Community Experience Distilled

Mastering Node.js

Expert techniques for building fast servers and scalable, real-time network applications with minimal effort

Sandro Pasquali

[PACKT] PUBLISHING open source* community experience distilled

Mastering Node.js

Expert techniques for building fast servers and scalable, real-time network applications with minimal effort

Sandro Pasquali

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Mastering Node.js

First published: November 2013

Production Reference: 1191113

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-632-0

www.packtpub.com

Cover Image by Jarek Blaminsky (milak6@wp.pl)

Credits

Author
Sandro Pasquali

Reviewers
Alex Kolundzija
Abhijeet Sutar
Kevin Faaborg

Acquisition Editors
Edward Gordan
Gregory Wild

Lead Technical Editor
Sweny M. Sukumaran

Technical Editors
Tanvi Bhatt
Jalasha D'costa
Akashdeep Kundu
Nikhil Potdukhe
Tarunveer Shetty
Sonali Vernekar

Project Coordinator
Kranti Berde

Proofreader
Amy Johnson

Indexer
Hemangini Bari

Graphics
Valentina D'Silva
Disha Haria
Yuvraj Manari

Production Coordinator
Kirtee Shingan

Cover Work
Kirtee Shingan

About the Author

Sandro Pasquali began writing games on a Commodore PET in grade school, and hasn't looked back. A polyglot programmer, who started with BASIC and assembly, his journey through C, Perl, and PHP led to JavaScript and the browser in 1995. He was immediately hooked on a vision of browsers as the software delivery mechanism of the future. By 1997 he had formed `Simple.com`, a technology company selling the world's first JavaScript-based application development framework, patenting several technologies and techniques that have proven prescient. Node represents for him only the natural next step in an inevitable march towards the day when all software implementations, and software users, are joined within a collaborative information network.

He has led the design of enterprise-grade applications for some of the largest companies in the world, including Nintendo, Major League Baseball, Bang and Olufsen, LimeWire, and others. He has displayed interactive media exhibits during the Venice Biennial, won design awards, built knowledge management tools for research institutes and schools, and has started and run several startups. Always seeking new ways to blend design excellence and technical innovation, he has made significant contributions across all levels of software architecture, from data management and storage tools to innovative user interfaces and frameworks.

He now works to mentor a new generation of developers also bitten by the collaborative software bug, especially the rabid ones.

Acknowledgments

Many people are responsible for the writing of this book. The team at Packt is owed many thanks for their diligent editing and guidance, not to mention their patience as my work evolved…slowly. Several dear colleagues and friends contributed ideas, feedback, and support. Heartfelt thanks go out to Kevin Faaborg, Michael Nutt, and Ernie Yu, whose insights regarding technology, software, society, and of course Node.js were invaluable in guiding me through the development of this book, and of my work in general. The reinforcing encouragement of Dre Labre, Stuart McDonald, David Furfero, John Politowski, Andy Ross, Alex Gallafent, Paul Griffin, Diana Barnes-Brown, and the others who listened politely while I thought out loud will remain with me as fond memories of this long process. I thank Joseph Werle for his energy and commitment, which was of great help as I grappled with some of the more obscure nuances of the Node.js platform.

In particular I would like to thank Alexander Kolundzija, whose early advocacy began this process, and who is, as T.S. Eliot once said of Ezra Pound, "il miglior fabbro".

The writing of this book kept me away from my family and friends for many days and nights, so I thank them all for putting up with my absences. Most importantly, to my darling wife Elizabeth, who faithfully supported me throughout, I send my love.

About the Reviewers

Kevin Faaborg is a professional software developer and avid software hobbyist. Along with JavaScript and Node.js, his work and interests include event-driven programming, open source software development, and peer-to-peer technology.

Alex Kolundzija is a full stack web developer with over a decade of experience at companies including Google, Meebo, and MLB.com. He's the founder and principal developer of Blend.io, a music collaboration network built with Node.js and a part of the Betaworks Studio of companies.

He has previously reviewed Kito Mann's *Java Server Faces in Action* (Manning).

Abhijeet Sutar is a computer science graduate from Mumbai University. He is a self-taught software developer, and enthusiastic about learning new technologies. His goto language is Java. He has mainly worked on middleware telephony applications for contact centers. He has also successfully implemented a highly available data store with MongoDB NoSQL database for a contact center application. He is currently moving onto Node.js platform for development of the next generation Operational Technology (OT). He blogs at `http://blog.ajduke.in`, codes at `http://github.com/ajduke` and tweets via handle `@_ajduke`.

I would like to thank the people at Packt Publishing, Krunal, Sweny, for providing reviewing opportunity for new technology, Node. I also want to thank Kranti for providing the chapters and putting reminders on due date, and promptly providing necessary information.

---

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

The Internet is no longer a collection of static websites to be passively consumed. The browser user has come to expect a much richer, interactive experience. Over the last decade or so, network applications have come to resemble desktop applications. Also, recognition of the social characteristics of information has inspired the development of new kinds of interfaces and visualizations modeling dynamic network states, where the user is viewing change over real time rather than fading snapshots trapped in the past.

Even though our expectations for software have changed, the tools available to us as software developers developers have not changed much. Computers are faster, and multicore chip architectures are common. Data storage is cheaper, as is bandwidth. Yet we continue to develop with tools designed before billion-user websites and push-button management of cloud-based clusters of virtual machines.

The development of network applications remains an overly expensive and slow process because of this. Developers use different languages, programming styles, complicating code maintenance, debugging, and more. Too regularly, scaling issues arrive too early, overwhelming the ability of what is often a small and inexperienced team. Popular modern software features, such as real-time data, multiplayer games, and collaborative editing spaces, demand systems capable of carrying thousands of simultaneous connections without bending. Yet we remain restricted to frameworks designed to assist us in building CRUD applications binding a single relational database on a single server to a single user running a multipage website in a browser on a desktop computer.

Node helps developers build more resilient network applications at scale. Built on C++ and bundled with Google's V8 engine, Node is fast, and it understands JavaScript. Node has brought together the most popular programming language in the world and the fastest JavaScript compiler around, and has given that team easy access to an operating system through C++ bindings. Node represents a change in how network software is designed and built.

What this book covers

Chapter 1, *Understanding the Node Environment*, gives a brief description of the particular problems Node attempts to solve, with a focus on how its single-threaded event-loop is designed, implemented, and used. We will also learn about how Google's V8 engine can be configured and managed, as well as best practices when building Node programs.

Chapter 2, *Understanding Asynchronous Event-Driven Programming*, digs deep into the fundamental characteristic of Node's design: event-driven, asynchronous programming. By the end of this chapter you will understand how events, callbacks, and timers are used in Node, as well as how the event loop works to enable high-speed I/O across filesystems, networks, and processes.

Chapter 3, *Streaming Data Across Nodes and Clients,* describes how streams of I/O data are knitted through most network software, emitted by file servers or broadcast in response to an HTTP GET request. Here we learn how Node facilitates the design, implementation, and composition of network software, using examples of HTTP servers, readable and writable file streams, and other I/O focused Node modules and patterns.

Chapter 4, *Using Node to Access the Filesystem*, lays out what you need to know when accessing the filesystem with Node, along with techniques for handling file uploads and other networked file operations.

Chapter 5, *Managing Many Simultaneous Client Connections*, shows you how Node helps in solving problems accompanying the high volume, high concurrency environments that contemporary, collaborative web applications demand. Through examples, learn how to efficiently track user state, route HTTP requests, handle sessions, and authenticate requests using the Redis database and Express web application framework.

Chapter 6, *Creating Real-Time Applications*, explores AJAX, Server-Sent-Events, and the WebSocket protocol, discussing their pros and cons, and how to implement each using Node. We finish the chapter by building a collaborative document editing application.

Chapter 7, *Utilizing Multiple Processes*, teaches how to distribute clusters of Node processes across multi-core processors, and other techniques for scaling Node applications. An investigation of the differences between programming in single and multithreaded environments leads to a discussion of how to spawn, fork, and communicate with child processes in Node, and we build an analytics tool that records, and displays, the mouse actions of multiple, simultaneous clients connected through a cluster of web sockets.

Chapter 8, *Scaling Your Application*, outlines some techniques for detecting when to scale, deciding how to scale, and scaling Node applications across multiple servers and cloud services, with examples including: how to use RabbitMQ as a message queue, using NGINX to proxy Node servers, and using Amazon Web Services in your application.

Chapter 9, *Testing Your Application*, explains how to implement unit, functional, and integration tests with Node. We will explore several testing libraries, including native Node assertion, sandboxing, and debugging modules. Examples using Grunt, Mocha, PhantomJS, and other build and testing tools accompany the discussion.

Appendix A, *Organizing Your Work*, gives tips on using the npm package management system. Learn how create packages, publish packages, and manage packages.

Appendix B, *Introducing the Path Framework*, demonstrates how to use this powerful full-stack application framework to build your next web application using only JavaScript, thanks to Node and its ability to handle thousands of simultaneously connected clients.

Appendix C, *Creating Your Own C++ Add-ons*, provides a brief introduction on how to build your own C++ add-ons, and how to use them from within Node.

What you need for this book

You will need to have some familiarity with JavaScript, and have a copy of Node installed on your development machine or server, Version 0.10.21 or higher. You should know how to install programs on this machine, as you will need to install Redis, along with other libraries, like PhantomJS. Having Git installed, and learning how to clone GitHub repositories, will greatly improve your experience.

You should install RabbitMQ so that you can follow with the examples using message queues. The sections on using NGINX to proxy Node servers will of course require that you can install and use that web server. To build C++ add-ons you will need to install the appropriate compiler on your system.

The examples in this book are built and tested within UNIX-based environments (including Mac OS X), but you should be able to run all Node examples on Windows-based operating systems as well. You can obtain installers for your system, and binaries, from `http://www.nodejs.org`.

Who this book is for

This book is for developers who want to build high-capacity network applications, such as social networks, collaborative document editing environments, real time data-driven web interfaces, networked games, and other I/O-heavy software. If you're a client-side JavaScript developer, reading this book will teach you how to become a server-side programmer using a language you already know. If you're a C++ hacker, Node is an open-source project built using that language, offering you an excellent opportunity to make a real impact within a large and growing community, even gaining fame, by helping to develop this exciting new technology.

This book is also for technical managers and others seeking an explanation of the capabilities and design philosophy of Node. The book is filled with examples of how Node solves the problems modern software companies are facing in terms of high-concurrency, real-time applications pushing enormous volumes of data through growing networks. Node has already been embraced by the enterprise, and you should consider it for your next project.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "To import modules into your Node program use the `require` directive."

A block of code is set as follows:

```
var EventEmitter = require('events').EventEmitter;
var Counter = function(init) {
  this.increment = function() {
      init++;
      this.emit('incremented', init);
  }
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
var size = process.argv[2];
var totl = process.argv[3] || 100;
var buff = [];
for(var i=0; i < totl; i++) {
```

```
    buff.push(new Buffer(size));
    process.stdout.write(process.memoryUsage().heapTotal + "\n");
  }
```

Any command-line input or output is written as follows:

```
> node process.js 1000000 100 > out.file
```

New terms and **important words** are shown in bold.

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Understanding the Node Environment

Node's goal is to provide an easy way to build scalable network programs.

– Ryan Dahl, creator of Node.js

The **WWW** (**World Wide Web**) makes it possible for hypermedia objects on the Internet to interconnect, communicating through a standard set of Internet protocols, commonly **HTTP** (**Hyper Text Transfer Protocol**). The growth in the complexity, number, and type of web applications delivering curated collections of these objects through the browser has increased interest in technologies that aid in the construction and management of intricate networked applications. Node is one such technology. By mastering Node you are learning how to build the next generation of software.

The hold that any one person has on information is tenuous. Complexity follows scale; confusion follows complexity. As resolution blurs, errors happen.

Similarly, the activity graph describing all expected **I/O** (**Input/Output**) interactions an application may potentially form between clients and providers must be carefully planned and managed, lest the capacity of both the system and its creator be overwhelmed. This involves controlling two dimensions of information: volume and shape.

As a network application scales, the volume of information it must recognize, organize, and maintain increases. This volume, in terms of I/O streams, memory usage, and **CPU** (**Central Processing Unit**) load, expands as more clients connect, and even as they leave (in terms of persisting user-specific data).

This expansion of information volume also burdens the application developer, or team of developers. Scaling issues begin to present themselves, usually demonstrating a failure to accurately predict the behavior of large systems from the behavior of small systems. Can a data layer designed for storing a few thousand records accommodate a few million? Are the algorithms used to search a handful of records efficient enough to search many more? Can this server handle 10,000 simultaneous client connections? The edge of innovation is sharp and cuts quickly, presenting less time for deliberation precisely when the cost of error is being magnified. The shape of objects comprising the whole of an application becomes amorphous and difficult to understand, particularly as ad hoc modifications are made, reactively, in response to dynamic tension in the system. What is described in a specification as a small subsystem may have been patched into so many other systems that its actual boundaries are misunderstood. It becomes impossible to accurately trace the outline of the composite parts of the whole.

Eventually an application becomes unpredictable. It is dangerous when one cannot predict all future states of an application, or the side effects of change. Any number of servers, programming languages, hardware architectures, management styles, and so on, have attempted to subdue the intractable problem of risk following growth, of failure menacing success. Oftentimes systems of even greater complexity are sold as the cure.

Node chose clarity and simplicity instead. There is one thread, bound to an event loop. Deferred tasks are encapsulated, entering and exiting the execution context via callbacks. I/O operations generate evented data streams, these piped through a single stack. Concurrency is managed by the system, abstracting away thread pools and simplifying memory management. Dependencies and libraries are introduced through a package management system, neatly encapsulated, and easy to distribute, install, and invoke.

Experienced developers have all struggled with the problems that Node aims to solve:

- How to serve many thousands of simultaneous clients efficiently
- Scaling networked applications beyond a single server
- Preventing I/O operations from becoming bottlenecks
- Eliminating single points of failure, thereby ensuring reliability
- Achieving parallelism safely and predictably

As each year passes, we see collaborative applications and software responsible for managing levels of concurrency that would have been considered rare just a few years ago. Managing concurrency, both in terms of connection handling and application design, is the key to building scalable web architectures.

In this book we will study the techniques professional Node developers use to tackle these problems. In this chapter, we will explore how a Node application is designed, the shape and texture of its footprint on a server, and the powerful base set of tools and features Node provides for developers. Throughout we will examine progressively more intricate examples demonstrating how Node's simple, comprehensive, and consistent architecture solves many difficult problems well.

Extending JavaScript

When he designed Node, JavaScript was not Ryan Dahl's original language choice. Yet, after exploring it, he found a very good modern language without opinions on streams, the filesystem, handling binary objects, processes, networking, and other capabilities one would expect to exist in a system's programming language. JavaScript, strictly limited to the browser, had no use for, and had not implemented, these features.

Dahl was guided by a few rigid principles:

- A Node program/process runs on a single thread, ordering execution through an event loop
- Web applications are I/O intensive, so the focus should be on making I/O fast
- Program flow is always directed through asynchronous callbacks
- Expensive CPU operations should be split off into separate parallel processes, emitting events as results arrive
- Complex programs should be assembled from simpler programs

The general principle is, operations must never block. Node's desire for speed (high concurrency) and efficiency (minimal resource usage) demands the reduction of waste. A waiting process is a wasteful process, especially when waiting for I/O.

JavaScript's asynchronous, event-driven design fits neatly into this model. Applications express interest in some future event and are notified when that event occurs. This common JavaScript pattern should be familiar to you:

```
Window.onload = function() {
  // When all requested document resources are loaded,
  // do something with the resulting environment
}

element.onclick = function() {
  // Do something when the user clicks on this element
}
```

The time it will take for an I/O action to complete is unknown, so the pattern is to ask for notification when an I/O event is emitted, whenever that may be, allowing other operations to be completed in the meantime.

Node adds an enormous amount of new functionality to JavaScript. Primarily, the additions provide evented I/O libraries offering the developer system access not available to browser-based JavaScript, such as writing to the filesystem or opening another system process. Additionally, the environment is designed to be modular, allowing complex programs to be assembled out of smaller and simpler components.

Let's look at how Node imported JavaScript's event model, extended it, and used it in the creation of interfaces to powerful system commands.

Events

Many of the JavaScript extensions in Node emit events. These events are instances of `events.EventEmitter`. Any object can extend `EventEmitter`, providing the developer with an elegant toolkit for building tight asynchronous interfaces to object methods.

Work through this example demonstrating how to set an `EventEmitter` object as the prototype of a function constructor. As each constructed instance now has the `EventEmitter` object exposed to its prototype chain, `this` provides a natural reference to the event **API** (**Application Programming Interface**). The `counter` instance methods can therefore emit events, and these can be listened for. Here we emit the latest count whenever the `counter.increment` method is called, and bind a callback to the incremented event, which simply prints the current counter value to the command line:

```
var EventEmitter = require('events').EventEmitter;
var Counter = function(init) {
  this.increment = function() {
      init++;
      this.emit('incremented', init);
  }
}
Counter.prototype = new EventEmitter();
var counter = new Counter(10);
var callback = function(count) {
    console.log(count);
}
counter.addListener('incremented', callback);

counter.increment(); // 11
counter.increment(); // 12
```

To remove the event listeners bound to `counter`, use `counter.removeListener('incremented', callback)`. For consistency with browser-based JavaScript, `counter.on` and `counter.addListener` are interchangeable.

The addition of `EventEmitter` as an extensible object greatly increases the possibilities of JavaScript on the server. In particular, it allows I/O data streams to be handled in an event-oriented manner, in keeping with the Node's principle of asynchronous, non-blocking programming:

```
var Readable = require('stream').Readable;
var readable = new Readable;
var count = 0;

readable._read = function() {
  if(++count > 10) {
    return readable.push(null);
  }
  setTimeout(function() {
    readable.push(count + "\n");
  }, 500);
};
readable.pipe(process.stdout);
```

In this program we are creating a `Readable` stream and piping any data pushed into this stream to `process.stdout`. Every 500 milliseconds we increment a counter and `push` that number (adding a newline) onto the stream, resulting in an incrementing series of numbers being written to the terminal. When our series has reached its limit (10), we push `null` onto the stream, causing it to terminate. Don't worry if you don't fully understand how `Readable` is implemented here—streams will be fully explained in the following chapters. Simply note how the act of pushing data onto a stream causes a corresponding event to fire, how the developer can assign a custom callback to handle this event, and how newly added data can be redirected to other streams. Node is designed such that I/O operations are consistently implemented as asynchronous, evented data streams.

It is also important to note the importance of this style of I/O. Because Node's event loop need only commit resources to handling callbacks, many other instructions can be processed in the down time between each interval.

As an exercise, re-implement the previous code snippet such that the emitted data is piped to a file. You'll need to use `fs.createWriteStream`:

```
var fs = require('fs');
var writeStream = fs.createWriteStream("./counter", {
  flags : 'w',
  mode: 0666
});
```

Modularity

In his book *The Art of Unix Programming*, *Eric Raymond* proposed the **Rule of Modularity**:

> *Developers should build a program out of simple parts connected by well defined interfaces, so problems are local, and parts of the program can be replaced in future versions to support new features. This rule aims to save time on debugging complex code that is complex, long, and unreadable.*

This idea of building complex systems out of small pieces, loosely joined is seen in the management theory, theories of government, physical manufacturing, and many other contexts. In terms of software development, it advises developers to contribute only the simplest and most useful component necessary within a larger system. Large systems are hard to reason about, especially if the boundaries of its components are fuzzy.

One of the primary difficulties when constructing scalable JavaScript programs is the lack of a standard interface for assembling a coherent program out of many smaller ones. For example, a typical web application might load dependencies using a sequence of `<script>` tags in the `<head>` section of an HTML document:

```
<head>
<script src="fileA.js"></script>
<script src="fileB.js"></script>
</head>
```

There are many problems with this sort of solution:

1. All potential dependencies must be declared prior to being needed—dynamic inclusion requires complicated hacks.
2. The introduced scripts are not forcibly encapsulated—nothing stops code in both files from writing to the same global object. Namespaces can easily collide, making arbitrary injection dangerous.
3. `fileA` cannot address `fileB` as a collection—an addressable context such as `fileB.method` isn't available.
4. The `<script>` method itself isn't systematic, precluding the design of useful module services, such as dependency awareness or version control.
5. Scripts cannot be easily removed, or overridden.
6. Because of these dangers and difficulties, sharing is not effortless, diminishing opportunities for collaboration in an open ecosystem.

Ambivalently inserting unpredictable code fragments into an application frustrates attempts to predictably shape functionality. What is needed is a standard way to load and share discreet program modules.

Accordingly, Node introduced the concept of the **package**, following the CommonJS specification. A package is a collection of program files bundled with a manifest file describing the collection. Dependencies, authorship, purpose, structure, and other important meta-data are exposed in a standard way. This encourages the construction of large systems from many small, interdependent systems. Perhaps even more importantly, it encourages sharing:

> *What I'm describing here is not a technical problem. It's a matter of people getting together and making a decision to step forward and start building up something bigger and cooler together.*
>
> — *Kevin Dangoor*, creator of CommonJS

In many ways the success of Node is due to growth in the number and quality of packages available to the developer community, distributed via Node's package management system, **npm**. The design choices of this system, both social and technical, have done much to help make JavaScript a viable professional option for systems programming.

More extensive information on creating and managing Node packages can be found in *Appendix A*, *Organizing Your Work*. The key point is this: build programs out of packages where possible, and share those packages when possible. The shape of your applications will be clearer and easier to maintain. Importantly, the efforts of thousands of other developers can be linked into applications via npm, directly by inclusion, and indirectly as shared packages are tested, improved, refactored, and repurposed by members of the Node community.

Contrary to popular belief, npm is not an abbreviation for **Node Package Manager** (or even an acronym):

`https://npmjs.org/doc/faq.html#If-npm-is-an-acronym-why-is-it-never-capitalized`

The Network

I/O in the browser is mercilessly hobbled, for very good reasons—if the JavaScript on any given website could access your filesystem, or open up network connections to any server, the WWW would be a less fun place.

For Node, I/O is of fundamental importance, and its focus from the start was to simplify the creation of scalable systems with high I/O requirements. It is likely that your first experience with Node was in writing an HTTP server.

Node supports several standard network protocols in addition to HTTP, such as **TLS/SSL** (**Transport Layer Security/Secure Sockets Layer**), and **UDP** (**User Datagram Protocol**). With these tools we can easily build scalable network programs, moving well beyond the somewhat dated **AJAX** (**Asynchronous JavaScript And Xml**) techniques familiar to the JavaScript developer.

Let's create a simple program that allows the user to send data between two UDP servers:

```
var dgram = require('dgram');
var client = dgram.createSocket("udp4");
var server = dgram.createSocket("udp4");

var message = process.argv[2] || "message";

message = new Buffer(message);

server
.on("message", function (msg) {
  process.stdout.write("Got message: " + msg + "\n");
  process.exit();

})
.bind(41234);

client.send(message, 0, message.length, 41234, "localhost");
```

Assuming a program file name of `udp.js` a message can be sent via UDP by running this program from the terminal like so:

```
node udp.js "my message"
```

Which will result in the following output:

```
Got message: my message
```

We first establish our UDP servers, one working as a broadcaster, the other as a listener. `process.argv` contains useful command information, including command-line arguments commencing at index(2), which in this case would contain `"my message"`. UDP requires messages to be `Buffer` objects, so we ensure that some message exists and convert it.

A UDP server is an instance of `EventEmitter`, emitting a message event when messages are received on the port it is bound. This server simply echoes the received message. All that is left to do is send the message, which action is performed by the client, passing along our message to port `#41234`.

Moving streams of data around the I/O layer of your application is simplified within Node. It isn't difficult to share data streams across differing protocol servers, as data streams are standardized via Node's interfaces. Protocol details are handled for you.

Let's continue to explore I/O, the `process` object, and events. First, let's dig into the machine powering Node's core.

V8

V8 is Google's JavaScript engine, written in C++. It compiles and executes JavaScript code inside of a **VM** (**Virtual Machine**). When a webpage loaded into Google Chrome demonstrates some sort of dynamic effect, like automatically updating a list or news feed, you are seeing JavaScript, compiled by V8, at work.

While Node itself will efficiently manage I/O operations, its `process` object refers to the V8 runtime. As such, it is important to understand how to configure the V8 environment, especially as your application grows in size.

By typing `node -h` into a console, something like the following will be displayed:

```
Options:
  -v, --version         print node's version
  -e, --eval script     evaluate script
  -p, --print           print result of --eval
  -i, --interactive     always enter the REPL even if stdin
                        does not appear to be a terminal
  --no-deprecation      silence deprecation warnings
  --trace-deprecation   show stack traces on deprecations
  --v8-options          print v8 command line options
  --max-stack-size=val  set max v8 stack size (bytes)
```

We can see how a list of V8 options is accessible via the `--v8-options` flag.

The list of configuration options for V8 is a long one, so we're not going to cover each option here. As we progress through the book, relevant options will be discussed with more depth. It is nevertheless useful to summarize some of the options provided for managing system limits and memory, as well as those used to configure JavaScript's command set, introducing some of the new features in **ES6** (**EcmaScript6**), often referred to as **Harmony**.

The version of V8 used by your Node installation can be viewed by typing:

```
node -e "console.log(process.versions.v8)"
```

Memory and other limits

One very powerful V8 configuration option is important enough to make it into Node's own collection: `--max-stack-size`. Let's look into some of the new powers a Node developer has been given in being able to configure a specific JavaScript runtime.

Trying to break a system is an excellent way to discover its limits and shape. Let's write a program that will crash V8:

```
var count = 0;
(function curse() {
  console.log(++count);
  curse();
})()
```

This self-contained, self-executing function will recursively call itself forever, or until it is forced to stop. Each iteration of `curse` adds another frame to the call stack. This uncontrolled growth will eventually cause the JavaScript runtime to collapse, citing a `RangeError: Maximum call stack size exceeded`.

The purpose of `--max-stack-size` should now be clear. The direct V8 option equivalent is `--stack_size`, which is passed a value, in **KB** (**Kilobytes**), to raise this limit. Experiment with the above program, noting the number of iterations possible at different settings.

While it is likely that hitting this limit represents an incorrectly designed algorithm, being able to expand the width of the operating space available to a Node process adds to the collection of solutions available to developers.

On 32 bit and 64 bit machines V8's memory allocation defaults are, respectively, 700 MB and 1400 MB. In newer versions of V8, memory limits on 64 bit systems are no longer set by V8, theoretically indicating no limit. However, the **OS** (**Operating System**) on which Node is running can always limit the amount of memory V8 can take, so the true limit of any given process cannot be generally stated.

V8 makes available the `--max_old_space_size` option, which allows control over the amount of memory available to a process, accepting a value in MB. Should you need to increase memory allocation, simply pass this option the desired value when spawning a Node process.

It is often an excellent strategy to reduce the available memory allocation for a given Node instance, especially when running many instances. As with stack limits, consider whether massive memory needs are better delegated to a dedicated storage layer, such as an in-memory database or similar.

An informative discussion with the V8 team regarding their views on how memory should be allocated can be found here:

`http://code.google.com/p/v8/issues/detail?id=847`

One of the key advantages of modern high-level languages such as JavaScript is the automatic management of memory through **GC** (**Garbage Collection**). GC strategies are many and complex, yet all follow a simple core idea: every so often, free allocated memory that is no longer being used.

The drawback of automatic GC is that it puts a slight brake on process speed. While the clear advantages of automatic GC outweigh its drawbacks in the majority of cases, there remains for the Node developer an opportunity to control some of its behavior. This is primarily done via the flags `--nouse_idle_notification` and `--expose_gc`.

Passing the `--nouse_idle_notification` flag will tell V8 to ignore idle notification calls from Node, which are requests to V8 asking it to run GC immediately, as the Node process is currently idle. Because Node is aggressive with these calls (efficiency breeds clean slates), an excess of GC may slow down your application. Note that using this flag does not disable GC; GC simply runs less often. In the right circumstances this technique can increase performance.

`--expose_gc` introduces a new global method to the Node process, `gc()`, which allows JavaScript code to manually start the GC process. In conjunction with `--nouse_idle_notification` the developer can now control to some degree how often GC runs. At any point in my JavaScript code I can simply call `gc()` and start the collector.

Being able to adjust memory usage and GC is certainly useful. Remember that application volume can rapidly rise in unpredictable ways. If the memory footprint of your application holds steady near the very limits of V8's allocation, you should begin to think about scaling horizontally. Use memory wisely, and split off new Node instances where appropriate.

Harmony

JavaScript has never stopped evolving and it is now experiencing something of a renaissance, helped in no small part by the popularity of Node. The language's next version, named Harmony, introduces some significant new features and concepts.

More information on ES6 Harmony can be found at:
`http://wiki.ecmascript.org/doku.php?id=harmony:harmony`.

The available Harmony options are:

Flag	Description
`--harmony_typeof`	Enable semantics for typeof
`--harmony_scoping`	Enable block scoping
`--harmony_modules`	Enable modules (implies block scoping)
`--harmony_proxies`	Enable proxies
`--harmony_collections`	Enable collections (sets, maps, and weak maps)
`--harmony`	Enable all features (except typeof)

It is beyond the scope of this book to discuss these new features in any depth. Nonetheless, it should be stated that this ability to use the very latest JavaScript features, now, through Node, offers the developer a great advantage—there is no browser war on a server to hold back innovation.

For example, ES6's `weakMap` allows the use of non-strings as keys in a HashMap:

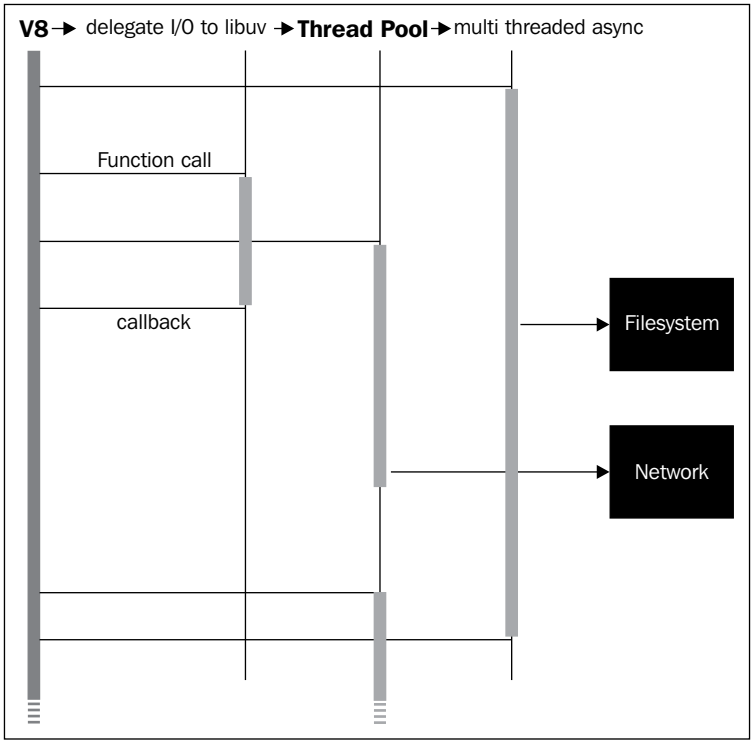
```
"use strict"
let owners = new WeakMap();
let task = {
  title   : "Big Project"
};
owners.set(task, 'John');
function owner(task) {
  if(owners.has(task)) {
    return console.log(owners.get(task));
  }
  console.log("No owner for this task.");
}
owner(task);   // "John"
owner({});   // "No owner for this task"
```

As an exercise, the reader might map (fixed) input streams to (variable) output streams in a similar manner.

The process object

By now it should be clear as to how Node is structured, in terms of V8, the event loop, and so forth. We are now going to discuss, in detail, how instructions that you write (a JavaScript program) are compiled by V8 into a list of instructions whose execution context is accessible via the native Node `process` object.

The single thread forming the spine of Node's event loop is V8's event loop. When I/O operations are initiated within this loop they are delegated to **libuv**, which manages the request using its own (multi-threaded, asynchronous) environment. libuv announces the completion of I/O operations, allowing any callbacks waiting on this event to be re-introduced to the main V8 thread for execution:



Node's `process` object provides information on and control over the current running process. It is an instance of `EventEmitter`, is accessible from any scope, and exposes very useful low-level pointers. Consider the following program:

```
var size = process.argv[2];
var totl = process.argv[3] || 100;
var buff = [];
for(var i=0; i < totl; i++) {
  buff.push(new Buffer(size));
  process.stdout.write(process.memoryUsage().heapTotal + "\n");
}
```

Downloading the example code:

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Assuming the program file is named `process.js`, it would be executed like so:

```
> node process.js 1000000 100
```

This execution context first fetches the two command-line arguments via `process.argv`, builds a looping construct that grows memory usage depending on these arguments, and emits memory usage data as each new allocation is made. The program sends output to `stdout`, but could alternatively stream output to other processes, or even a file:

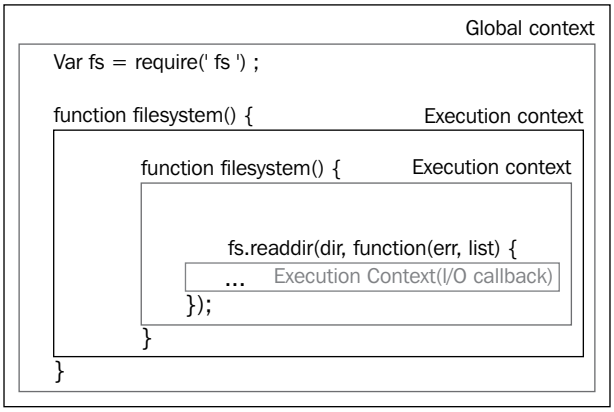
```
> node process.js 1000000 100 > out.file
```

In fact, the familiar `console.log` is implemented in the Node core as a wrapper around `process.stdout.write`:

```
console.log = function (d) {
  process.stdout.write(d + '\n');
};
```

A Node process begins by constructing a single execution stack, with the global context forming the base of the stack. Functions on this stack execute within their own, local, context (sometimes referred to as scope), which remains enclosed within the global context (which you'll hear referred to as closure). Because Node is evented, any given execution context can commit the running thread to handling an eventual execution context. This is the purpose of callback functions.

Consider the following schematic of a simple interface for accessing the filesystem:



If we were to instantiate `Filesystem` and call `readDir` a nested execution context structure would be created: (global (fileSystem (readDir (anonymous function)))). The concomitant execution stack is introduced to Node's single process thread. This stack remains in memory until libuv reports that `fs.readdir` has completed, at which point the registered anonymous callback fires, resolving the sole pending execution context. As no further events are pending, and the maintenance of closures no longer necessary, the entire structure can be safely torn down (in reverse, beginning with anonymous), and the process can exit, freeing any allocated memory. This method of building up and tearing down a single stack is what Node's event loop is ultimately doing.

We'll explore the full suite of commands and attributes contained by the `process` object as we continue to develop examples and libraries in this book.

The Read-Eval-Print Loop and executing a Node program

Node's **REPL** (**Read-Eval-Print-Loop**) represents the Node shell. To enter the shell prompt, enter Node via your terminal without passing a filename:

```
> node
```

You now have access to a running Node process, and may pass JavaScript commands to this process. For example, after entering `2+2` the shell would send `4` to `stdout`. Node's REPL is an excellent place to try out, debug, test, or otherwise play with JavaScript code.

Because the REPL is a native object, programs can also use instances as a context in which to run JavaScript interactively. For example, here we create our own custom function `sayHello`, add it to the context of a REPL instance, and start the REPL, emulating a Node shell prompt:

```
require('repl').start("> ").context.sayHello = function() {
    return "Hello"
};
```

Entering `sayHello()` at the prompt will result in `Hello` being sent to `stdout`.

Let's take everything we've learned in this chapter and create an interactive REPL which allows us to execute JavaScript on a remote server.

Create two files, `repl_client.js` and `repl_server.js`, using the following code, and run each in its own terminal window, such that both terminal windows are visible to you.

```
/* repl_client.js   */
var net = require('net');
var sock = net.connect(8080);
process.stdin.pipe(sock);
sock.pipe(process.stdout);


/* repl_server.js  */
var repl = require('repl')
var net = require('net')
net.createServer(function(socket) {
  repl
  .start({
    prompt : '> ',
    input    : socket,
    output  : socket,
    terminal  : true
  })
  .on('exit', function () {
    socket.end()
  })
}).listen(8080)
```

`repl_client` simply creates a new socket connection to port `8080` through `net.connect`, and pipes any data coming from `stdin` (your terminal) through that socket. Similarly, any data arriving from the socket is piped to `stdout` (your terminal).
It should be clear that we have created a way to take input and send it via a socket to port `8080`, listening for any data that the socket may send back to us.

`repl_server` closes the loop. We first create a new **TCP** (**Transmission Control Protocol**) server with `net.createServer`, binding to port 8080 via `.listen`. The callback passed to `net.createServer` will receive a reference to the bound socket. Within the enclosure of that callback we instantiate a new REPL instance, giving it a nice `prompt` (> here, but could be any string), indicating that it should both listen for `input` from, and broadcast `output` to, the passed socket reference, indicating that the socket data should be treated as `terminal` data (which has special encoding).

We can now type something like `console.log("hello")` into the client terminal, and see `hello` displayed.

To confirm that the execution of our JavaScript commands is occurring in the server instance, type `console.log(process.argv)` into the client, and the server will display an object containing the current process path, which will be `repl_server.js`.

It should be clear from this demonstration that we have created a way to remotely control Node processes. It is a short step from here to multi-node analytics tools, remote memory management, automatic server administration, and so forth.

Summary

In this chapter we've outlined the key problems Node's designers sought to solve, and how their solution has made the creation of easily scalable, high-concurrency networked systems easier for an open community of developers. We've seen how JavaScript has been given very useful new powers, how its evented model has been extended, and how V8 can be configured to further customize the JavaScript runtime.

Through examples, we've learned how I/O is handled by Node, how to program the REPL, as well as how to manage inputs and outputs to the process object. The goal of demonstrating how Node allows applications to be intelligently constructed out of well-formed pieces in a principled way has begun. In the next chapter, we will delve deeper into asynchronous programming, learn how to manage more complex event chains, and develop more powerful programs using Node's model.

2

Understanding Asynchronous Event-Driven Programming

The best way to predict the future is to invent it.

– *Alan Kay*

Eliminating blocking processes through the use of event-driven, asynchronous I/O is Node's primary organizational principle. We've learned how this design helps developers in shaping information and adding capacity: lightweight, independent, and share-nothing processes communicating through callbacks synchronized within a predictable event loop.

Accompanying the growth in the popularity of Node is a growth in the number of well-designed evented systems and applications. For a new technology to be successful, it must eliminate existing problems and/or offer to consumers a better solution at a lower cost in terms of time or effort or price. In its short and fertile lifespan, the Node community has collaboratively proven that this new development model is a viable alternative to existing technologies. The number and quality of Node-based solutions powering enterprise-level applications provide further proof that these new ideas are not only novel, but preferred.

In this chapter we will delve deeper into how Node implements event-driven programming. We will begin by unpacking the ideas and theories that event-driven languages and environments derive from and grapple with, in an effort to clear away misconceptions and encourage mastery. Following this introduction, more detail on how timers, callbacks, I/O events, flow control, and the event loop are implemented and used will be laid out. Theory will be practiced as we build up a simple but exemplary file and data-driven applications, highlighting Node's strengths, and how it is succeeding in its ambition to simplify network application designs.

Broadcasting events

It is always good to have an accurate understanding of the total eventual cost of asking for a service to be performed.

I/O is expensive. In the following chart (taken from *Ryan Dahl*'s original presentation on Node) we can see how many clock cycles typical system tasks consume. The relative cost of I/O operations is striking.

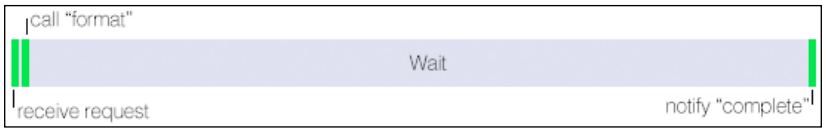
L1 cache	3 cycles
L2 cache	14 cycles
RAM	250 cycles
Disk	41,000,000 cycles
Network	240,000,000 cycles

The reasons are clear enough: a disk is a physical device, a spinning metal platter that buses data at a speed that cannot possibly match the speed of an on-chip or near-chip cache moving data between the CPU and RAM (Random Access Memory). Similarly, a network is bound by the speed in which data can travel through its connecting "wires", modulated by its controllers. Even through fiber optic cables, light itself needs 0.1344 seconds to travel around the world. In a network used by billions of people regularly interacting across great distances, this sort of latency builds up.

In the traditional marketplace described by an application running on a blocking system the purchase of a file operation requires a significant expenditure of resources, as we can see in the preceding table. Primarily this is due to scarcity: a fixed number of processes, or "units of labor" are available, each able to handle only a single task, and as the availability of labor decreases, its cost (to the client) increases.

The breakthrough in thinking reflected by Node's design is simple to understand once one recognizes that most worker threads spend their time waiting—for more instructions, a sub-task to complete, and so on. For example, a process assigned to service the command *format my hard drive* will dedicate all of its allotted resources to managing a workflow something like the following:

- Communicate to a device driver that a format request has been made
- Idle, waiting for an "unknowable" length of time
- Receive the signal *format is complete*
- Notify the client
- Clean up; shut down



In the preceding figure we see that an expensive worker is charging the client a fixed fee per unit of time regardless of whether any useful work is being done (the client is paying equally for activity and idleness). Or to put it another way, it is not necessarily true, and most often simply not true, that the sub-tasks comprising a total task each require identical effort or expertise, and therefore it is wasteful to pay a premium price for such cheap labor.

Sympathetically, we must also recognize that this worker can do no better even if ready and able to handle more work—even the best intentioned worker cannot do anything about I/O bottlenecks. The worker here is **I/O bound**.

A **blocking** process is therefore better understood as an **idle** process, and **idle** processes are **bottlenecks** within the particular task and for the overall application flow. What if multiple clients could share the same worker, such that the moment a worker announces availability due to an I/O bottleneck, another job from another client could be started?

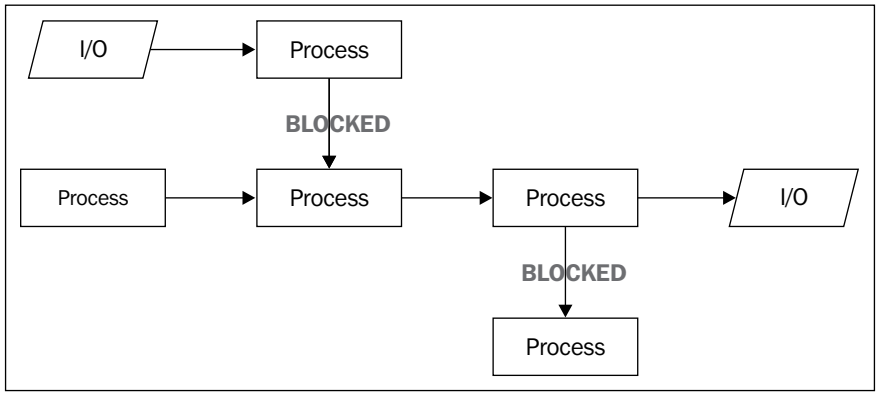
Node has commoditized I/O through the introduction of an environment where system resources are (ideally) never idle. Event-driven programming as implemented by Node reflects the simple goal of lowering overall system costs by encouraging the sharing of expensive labor, mainly by reducing the number of I/O bottlenecks to zero. We no longer have a powerless chunk of rigidly-priced unsophisticated labor; we can reduce all effort into discrete units with precisely delineated shapes and therefore admit much more accurate pricing. Identical outlays of capital can fund a much larger number of completed transactions, increasing the efficiency of the market *and* the potential of the market in terms of new products and new product categories. Many more concurrent transactions can be handled on the same infrastructure, at the same cost.

If the start, stop, and idle states of a process are understood as being **events** that can be subscribed to and acted upon we can begin to discuss how extremely complex systems can be constructed within this new, and at heart quite simple to grasp, model.

What would an environment within which many client jobs are cooperatively scheduled look like? And how is this message passing between events handled?

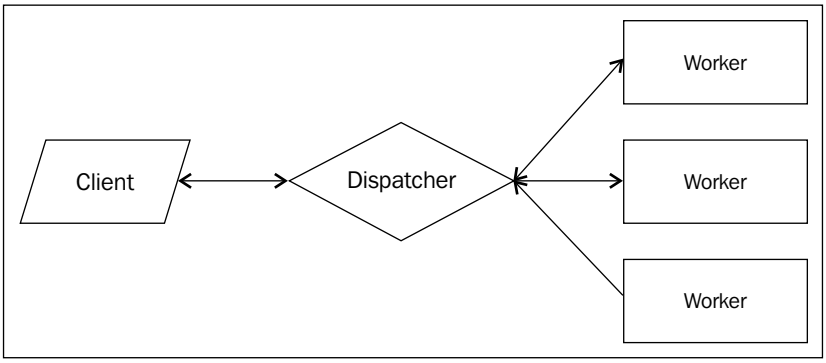
Collaboration

The worker flow described in the previous section is an example of a blocking server. Each worker is assigned a task or process, with each process able to accept only one request for work. They'll be blocking other requests, even if idling:



What would be preferable is a collaborative work environment, where workers could be assigned new tasks to do, instead of idling. In order to achieve such a goal what is needed is a virtual switchboard, where requests for services could be dispatched to available workers, and where workers could notify the switchboard of their availability.

One way to achieve this goal would be to maintain the idea of having a pool of available labors, but improving efficiency by delegating tasks to different workers as they come in:



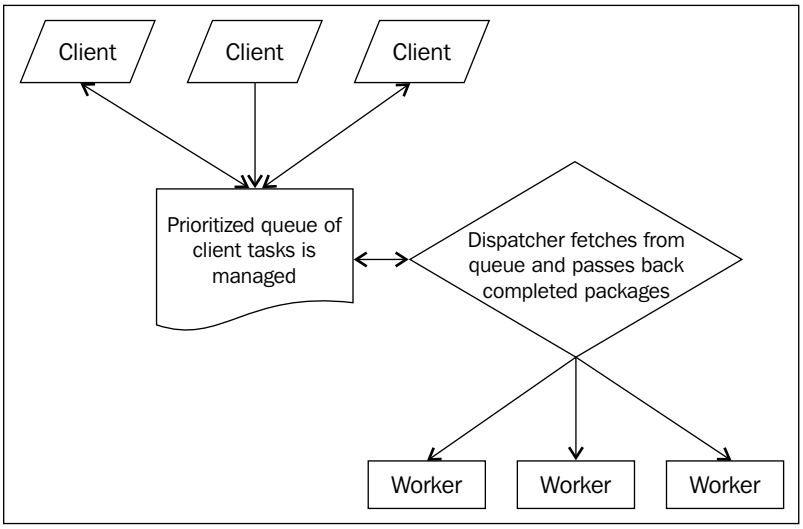
One drawback to this method is the amount of scheduling and worker surveillance that needs to be done. The dispatcher must field a continuous stream of requests, while managing messages coming from workers about their availability, neatly breaking up requests into manageable tasks and efficiently sorting them such that the fewest number of workers are idling.

Perhaps most importantly, what happens when all workers are fully booked? Does the dispatcher begin to drop requests from clients? Dispatching is resource-intensive as well, and there are limits even to the dispatcher's resources. If requests continue to arrive and no worker is available to service them what does the dispatcher do? Manage a queue? We now have a situation where the dispatcher is no longer doing the right job (dispatching), and has become responsible for bookkeeping and keeping lists, further diminishing operational efficiency.

Queueing

In order to avoid overwhelming anyone, we might add a buffer between the clients and the dispatcher.

This new worker is responsible for managing customer relations. Instead of speaking directly with the dispatcher, the client speaks to the services manager, passing the manager requests, and at some point in the future getting a call that their task has been completed. Requests for work are added to a prioritized work queue (a stack of orders with the most important one on top), and this manager waits for another client to walk through the door. The following figure describes the situations:



When a worker is idle the dispatcher can fetch the first item on the stack, pass along any package workers have completed, and generally maintain a sane work environment where nothing gets dropped or lost. If it comes to a point where all the workers are idle and the task queue is empty, the office can sleep for a while, until the next client arrives.

This last model inspires Node's design. The primary modification is to occupy the workers' pool solely with I/O tasks and delegate the remaining work to the single thread of V8. If a JavaScript program is understood as the client, Node is the services manager running through the provided instructions and prioritizing them. When a potentially blocking task is encountered (I/O, timers, and streams) it is handed over to the dispatcher (the libuv thread pool). Otherwise, the instruction is queued up for the event loop to pop and execute.

Listening for events

In the previous chapter we were introduced to the `EventEmitter` interface. This is the primary event interface we will be encountering as we move chapter to chapter, as it provides the prototype class for the many Node objects exposing evented interfaces, such as file and network streams. Various `close`, `exit`, `data`, and other events exposed by different module APIs signal the presence of an `EventEmitter` interface, and we will be learning about these modules and use cases as we progress.

Instead, the primary purpose of this section is to discuss some lesser-known **event sources**—signals, child process communication, filesystem change events, and deferred execution.

Signals

In many ways, evented programming is like hardware interrupt programming. Interrupts do exactly what their name suggests. They use their ability to interrupt whatever a controller or the CPU or any other device is doing, demanding that their particular need be serviced immediately.

In fact, the Node `process` object exposes standard **Portable Operating System Interface** (**POSIX**) signal names, such that a node process can subscribe to these system events.

> *A signal is a limited form of inter-process communication used in Unix, Unix-like, and other POSIX-compliant operating systems. It is an asynchronous notification sent to a process or to a specific thread within the same process in order to notify it of an event that occurred.*
>
> *—http://en.wikipedia.org/wiki/POSIX_signal*

This is a very elegant and natural way to expose a Node process to operating system (OS) signal events. One might configure listeners to catch signals instructing a Node process to restart or update some configuration files or simply clean up and shut down.

For example, the **SIGINT** signal is sent to a process when its controlling terminal detects a *Ctrl-C* (or equivalent) keystroke. This signal tells a process that an interrupt has been requested. If a Node process has bound a callback to this event, that function might log the request prior to terminating, do some other cleanup work, or even ignore the request:

```
setInterval(function() {}, 1e6);
process.on('SIGINT', function() {
    console.log('SIGINT signal received');
    process.exit(1);
})
```

Here we have set up a far future interval such that the process does not immediately terminate, and a SIGINT listener. When a user sends a *Ctrl-C* interrupt to the terminal controlling this process, the message **SIGINT signal received** will be written on the terminal and the process will terminate.

Now consider a situation in which a Node process is doing some ongoing work, such as parsing logs. It might be useful to be able to send that process a signal, such as *update your configuration files* or *restart the scan*. You may want to send such signals from the command line. You might prefer to have another process do so—a practice known as **Inter-Process Communication** (**IPC**).

Create a file named `ipc.js` containing the following code:

```
setInterval(function() {}, 1e6);
process.on('SIGUSR1', function() {
    console.log('Got a signal!');
});
```

`SIGUSR1` (and `SIGUSR2`) are user-defined signals (they are triggered by no specific action). This makes them ideal signals for custom functionality.

To send a command to a process you must determine its **process ID** (**PID**). With a PID in hand, processes can be addressed, and therefore communicated with. If the PID assigned to `ipc.js` after being run through Node is `123`, then we can send that process a `SIGUSR1` signal using the following command line:

```
kill -s SIGUSR1 123
```

A simple way to find the PID for a given Node process in UNIX is to search the system process list for the name of the program that says the process is running. If `ipc.js` is currently executing, its PID is found by entering the following command line in the console/terminal:

```
ps aux | grep ipc.js
```

Try it.

Forks

A fundamental part of Node's design is to create or fork processes when parallelizing execution or scaling a system—as opposed to creating a thread pool, for instance. We will be using child processes in various ways throughout this book, and learn how to create and use them. Here the focus will be on understanding how communication events between child processes are to be handled.

To create a child process one need simply call the `fork` method of the `child_process` module, passing it the name of a program file to execute within the new process:

```
var cp = require('child_process');
var child = cp.fork(__dirname + '/lovechild.js');
```

In this way any number of subprocesses can be kept running. Additionally, on multicore machines, forked processes will be distributed (by the OS) to different cores. Spreading node processes across cores (even other machines) and managing IPC is (one) way to scale a Node application in a stable, understandable, and predictable way.

Extending the preceding, we can now have the forking process (parent) send, and listen for, messages from the forked process (child):

```
child.on('message', function(msg) {
    console.log('Child said: ', msg);
});
child.send("I love you");
```

Similarly, the child process (its program is defined in `lovechild.js`) can send and listen for messages:

```
// lovechild.js
process.on('message', function(msg) {
    console.log('Parent said: ', msg);
    process.send("I love you too");
});
```

Running `parent.js` should fork a child process and send that child a message. The child should respond in kind:

```
Parent said:  I love you
Child said:  I love you too
```

Another very powerful idea is to pass a network server an object to a child. This technique allows multiple processes, including the parent, to share the responsibility for servicing connection requests, spreading load across cores.

For example, the following program will start a network server, fork a child process, and pass this child the server reference:

```
var child = require('child_process').fork('./child.js');
var server = require('net').createServer();
server.on('connection', function(socket) {
    socket.end('Parent handled connection');
});
server.listen(8080, function() {
    child.send("The parent message", server);
});
```

In addition to passing a message to a child process as the first argument to `send`, the preceding code also sends the server handle to itself as a second argument. Our child server can now help out with the family's service business:

```
//    child.js
process.on('message', function(msg, server) {
    console.log(msg);
    server.on('connection', function(socket) {
        socket.end('Child handled connection');
    });
});
```

This child process should print out the sent message to your console, and begin listening for connections, sharing the sent server handle. Repeatedly connecting to this server at `localhost:8080` will result in *either* **Child handled connection** or **Parent handled connection** being displayed; two separate processes are balancing the server load. It should be clear that this technique, when combined with the simple inter-process messaging protocol discussed previously, demonstrates how Ryan Dahl's creation succeeds in providing *an easy way to build scalable network programs*.

We will discuss Node's new `cluster` module, which expands (and simplifies) the previously discussed technique in later chapters. If you are interested in how server handles are shared, visit the `cluster` documentation at the following link:

`http://nodejs.org/api/cluster.html`

For those who are truly curious, examine the `cluster` code itself at:

`https://github.com/joyent/node/blob/c668185adde3a474585a11f172b8387e270ec23b/lib/cluster.js#L523-558`

File events

Most applications make some use of the filesystem, in particular those that function as web services. As well, a professional application will likely log information about usage, cache pre-rendered data views, or make other regular changes to files and directory structures.

Node allows developers to register for notifications on file events through the `fs.watch` method. The `watch` method will broadcast changed events on both files *and* directories.

`watch` accepts three arguments, in order:

1. The file or directory path being watched. If the file does not exist an **ENOENT** (**no entity**) error will be thrown, so using `fs.exists` at some prior useful point is encouraged.
2. An optional options object:
    - persistent (Boolean): Node keeps processes alive as long as there is "something to do". An active file watcher will by default function as a persistence flag to Node. Setting this option to false flags *not* keeping the general process alive if the watcher is the only activity keeping it running.
3. The listener function, which receives two arguments:
    - The name of the change event (one of *rename* or *change*).
    - The filename that was changed (important when watching directories).

Some operating systems will *not* return this argument.

This example will set up a watcher on itself, change its own filename, and exit:

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

fs.watch(__filename, { persistent: false }, function(event, filename)
{
    console.log(event);
    console.log(filename);
})

setImmediate(function() {
    fs.rename(__filename, __filename + '.new', function() {});
});
```

Two lines, `rename` and the name of the original file, should have been printed to the console.

Watcher channels can be closed at any time using the following code snippet:

```
var w = fs.watch('file', function(){})
w.close();
```

It should be noted that `fs.watch` depends a great deal on how the host OS handles file events, and according to the Node documentation:

> "*The* `fs.watch` *API is not 100% consistent across platforms, and is unavailable in some situations.*"

The author has had very good experiences with the module across many different systems, noting only that the filename argument is `null` in callbacks on OS X implementations. Nevertheless, be sure to run tests on your specific architecture—trust, but verify.

Deferred execution

One occasionally needs to defer the execution of a function. Traditional JavaScript uses timers for this purpose, the well-known `setTimeout` and `setInterval` functions. Node introduces another perspective on defers, primarily as means of controlling the order in which a callback executes in relation to I/O events, as well as timer events properly.

We'll learn more about this ordering in the event loop discussion that follows. For now we will examine two types of deferred event sources that give a developer the ability to schedule callback executions to occur either before, or after, the processing of queued I/O events.

process.nextTick

A method of the native Node `process` module, `process.nextTick` is similar to the familiar `setTimeout` method in which it delays execution of its callback function until some point in the future. However, the comparison is not exact; a list of all requested `nextTick` callbacks are placed at the head of the event queue and is processed, in its entirety and in order, *before* I/O or timer events and *after* execution of the current script (the JavaScript code executing synchronously on the V8 thread).

The primary use of `nextTick` in a function is to postpone the broadcast of result events to listeners on the current execution stack until the caller has had an opportunity to register event listeners—to give the currently executing program a chance to bind callbacks to `EventEmitter.emit` events. It may be thought of as a pattern used wherever asynchronous behavior should be emulated. For instance, imagine a lookup system that may either fetch from a cache or pull fresh data from a data store. The cache is fast and doesn't need callbacks, while the data I/O call would need them. The need for callbacks in the second case argues for emulation of the callback behavior with `nextTick` in the first case. This allows a consistent API, improving clarity of implementation without burdening the developer with the responsibility of determining whether or not to use a callback.

The following code seems to set up a simple transaction; when an instance of `EventEmitter` emits a `start` event, log **"Started"** to the console:

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

function getEmitter() {
    var emitter = new events.EventEmitter();
    emitter.emit('start');
    return emitter;
}

var myEmitter = getEmitter();

myEmitter.on("start", function() {
    console.log("Started");
});
```

However, the expected result will not occur. The event emitter instantiated within `getEmitter` emits `"start"` previous to being returned, wrong-footing the subsequent assignment of a listener, which arrives a step late, missing the event notification.

To solve this race condition we can use `process.nextTick`:

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

function getEmitter() {
```

```
    var emitter = new events.EventEmitter();
    process.nextTick(function() {
        emitter.emit('start');
    });
    return emitter;
}
var myEmitter = getEmitter();
myEmitter.on('start', function() {
    console.log('Started');
})
```

Here the attachment of the `on(start` handler is allowed to occur *prior* to the emission of the start event by the emitter instantiated in `getEmitter`.

Because it is possible to recursively call `nextTick`, which might lead to an infinite loop of recursive `nextTick` calls (starving the event loop, preventing I/O), there exists a failsafe mechanism in Node which limits the number of recursive `nextTick` calls evaluated prior to yielding the I/O: `process.maxTickDepth`. Set this value (which defaults to 1000) if such a construct becomes necessary—although what you probably want to use in such a case is `setImmediate`.

setImmediate

`setImmediate` is technically a member of the class of timers (`setInterval`, `setTimeout`). However, there is no sense of time associated with it—there is no *number of milliseconds to wait* argument to be sent. This method is really more of a sister to `process.nextTick`, differing in one very important way; while callbacks queued by `nextTick` will execute *before* I/O and timer events, callbacks queued by `setImmediate` will be called *after* I/O events.

The naming of these two methods is confusing: `nextTick` occurs **before** `setImmediate`.

This method does reflect the standard behavior of timers in that its invocation will return an object which can be passed to `cancelImmediate`, cancelling `setImmediate` in the same way `cancelTimeout` cancels timers set with `setTimeout`.

Timers

Timers are used to schedule events in the future. They are used when one seeks to delay the execution of some block of code until a specified number of milliseconds have passed, to schedule periodic execution of a particular function, or to slot some functionality immediately to the following.

JavaScript provides two asynchronous timers: `setInterval()` and `setTimeout()`.

It is assumed that the reader is fully aware of how to set (and cancel) these timers, so very little time will be spent discussing the syntax. We'll instead focus more on "gotchas" and "less well-known" details about timeouts and intervals.

The key takeaway will be this: when using timers one should make no assumptions about the amount of *actual* time that will expire before the callback registered for this timer fires, or about the ordering of callbacks. Node timers are *not* interrupts. Timers simply promise to execute as close as possible to the specified time (though never before), beholden, as with every other event source, to event loop scheduling.

At least one thing you may not know about timers...

We are all familiar with the standard arguments to `setTimeout`: a callback function and timeout interval. Did you know that many additional arguments are passed to the callback function?

```
setTimeout(callback, time, [passArg1, passArg2…])
```

setTimeout

Timeouts are used to defer the execution of a function until some number of milliseconds into the future:

Consider the following code:

```
setTimeout(a, 1000);
setTimeout(b, 1001);
```

One would expect that function `b` would execute after function `a`. However, this cannot be guaranteed—`a` may follow `b`, or the other way around.

Now, consider the subtle difference present in the following code snippet:

```
setTimeout(a, 1000);
setTimeout(b, 1000);
```

The execution order of `a` and `b` are predictable in this case. Node essentially maintains an object map grouping callbacks with identical timeout lengths. *Isaac Schlueter*, the current leader of the Node project, puts it this way:

> *[N]ode uses a single low level timer object for each timeout value. If you attach multiple callbacks for a single timeout value, they'll occur in order, because they're sitting in a queue. However, if they're on different timeout values, then they'll be using timers in different threads, and are thus subject to the vagaries of the [CPU] scheduler.*
>
> *—https://groups.google.com/forum/#!msg/nodejs-dev/kiowz4iht4Q/T0RuSwAeJV0J*

The ordering of timer callbacks registered within an identical execution scope does not predictably determine the eventual execution order in all cases.

Additionally, there exists a minimum wait time of one millisecond for a timeout. Passing a value of zero, -1, or a non-number will be translated into this minimum value.

setInterval

One can think of many cases where being able to periodically execute a function would be useful. Polling a data source every few seconds and pushing updates is a common pattern. Running the next step in an animation every few milliseconds is another use case, as is collecting garbage. For these cases `setInterval` is a good tool:

```
var intervalId = setInterval(function() { ... }, 100);
```

Every 100 milliseconds the sent callback function will execute, a process that can be cancelled with `clearInterval(intervalId)`.

Unfortunately, as with `setTimeout`, this behavior is not always reliable. Importantly, if a system delay (such as some badly written blocking `while` loop) occupies the event loop for some period of time, intervals set prior and completing within that interim will have their results queued on the stack. When the event loop becomes unblocked and unwinds, *all* the interval callbacks will be fired in sequence, essentially immediately, losing any sort of timing delays they intended.

Luckily, unlike browser-based JavaScript, intervals are rather more reliable in Node, generally able to maintain expected periodicity in normal use scenarios.

unref and ref

A Node program does not *stay alive* without a reason to do so. A process will keep running for as long as there are callbacks still waiting to be processed. Once those are cleared, the Node process has nothing left to do, and it will exit.

For example, the following silly code fragment will keep a Node process running forever:

```
Var intervalId = setInterval(function() {}, 1000);
```

Even though the set callback function does nothing useful or interesting, it continues to be called—and this is the correct behavior, as an interval should keep running until `clearInterval` is used to stop it.

There are cases of using a timer to do something interesting with external I/O, or some data structure, or a network interface where once those external event sources stop occurring or disappear, the timer itself stops being necessary. Normally one would trap that *irrelevant* state of a timer somewhere else in the program and cancel the timer from there. This can become difficult or even clumsy, as an unnecessary tangling of concerns is now necessary, an added level of complexity.

The `unref` method allows the developer to assert the following instructions: *when this timer is the only event source remaining for the event loop to process, go ahead and terminate the process*.

Let's test this functionality to our previous silly example, which will result in the process terminating rather than running forever:

```
var intervalId = setInterval(function() {}, 1000);
intervalId.unref();
```

Note that `unref` is a method of the opaque value returned when starting a timer (which is an object).

Now let's add an external event source, a timer. Once that external source gets cleaned up (in about 100 milliseconds), the process will terminate. We send information to the console to log what is happening:

```
setTimeout(function() {
    console.log("now stop");
}, 100);
var intervalId = setInterval(function() {
    console.log("running")
}, 1);
intervalId.unref();
```

You may return a timer to its normal behavior with `ref`, which will undo an `unref` method:

```
var intervalId = setInterval(function() {}, 1000);
intervalId.unref();
intervalId.ref();
```

The listed process will continue indefinitely, as in our original silly example.

Understanding the event loop

Node processes JavaScript instructions using a single thread. Within your JavaScript program no two operations will ever execute at exactly the same moment, as might happen in a multithreaded environment. Understanding this fact is essential to understanding how a Node program, or process, is designed and runs.

This does not mean that only one thread is being used on the machine hosting this a Node process. Simply writing a callback does not magically create parallelism! Recall *Chapter 1*, *Understanding the Node Environment*, and our discussion about the `process` object—Node's "single thread" simplicity is in fact an abstraction created for the benefit of developers. It is nevertheless crucial to remember that there are many threads running in the background managing I/O (and other things), and these threads unpredictably insert instructions, originally packaged as callbacks, into the single JavaScript thread for processing.

Node executes instructions one by one until there are no further instructions to execute, no more input or output to stream, and no further callbacks waiting to be handled.

Even deferred events (such as timeouts) require an eventual interrupt in the event loop to fulfill their promise.

For example, the following `while` loop will never terminate:

```
var stop = false;
setTimeout(function() {
    stop = true;
}, 1000);

while(stop === false) {};
```

Even though one might expect, in approximately one second, the assignment of a Boolean `true` to the variable `stop`, tripping the `while` conditional and interrupting its loop, this will *never happen*. Why? This `while` loop starves the event loop by running infinitely, greedily checking and rechecking a value that is never given a chance to change, as the event loop is never given a chance to schedule our timer callback for execution.

As such, programming Node implies programming the event loop. We've previously discussed the *event sources* that are queued and otherwise arranged and ordered on this event loop—I/O events, timer events, and so on.

When writing non-deterministic code it is imperative that no assumptions about eventual callback orders are made. The abstraction that is Node masks the complexity of the thread pool on which the straightforward main JavaScript thread floats, leading to some surprising results.

We will now refine this general understanding with more information about how, precisely, the callback execution order for each of these types is determined within Node's event loop.

Four sources of truth

We have learned about the four main groups of deferred event sources, whose position and priority on the stack we will now demonstrate:

- **Execution blocks**: The blocks of JavaScript code comprising the Node program, being expressions, loops, functions, and so on. This includes `EventEmitter` events emitted within the current execution context.
- **Timers**: Callbacks deferred to sometime in the future specified in milliseconds, such as `setTimeout` and `setInterval`.
- **I/O**: Prepared callbacks returned to the main thread after being delegated to Node's managed thread pool, such as filesystem calls and network listeners.
- **Deferred execution blocks**: Mainly the functions slotted on the stack according to the rules of `setImmediate` and `nextTick`.

We have learned how the deferred execution method `setImmediate` slots its callbacks *after* I/O callbacks in the event queue, and `nextTick` slots its callbacks *before* I/O and timer callbacks.

A challenge for the reader

After running the following code, what is the expected order of logged messages?

```
var fs = require('fs');
var EventEmitter = require('events').EventEmitter;
var pos = 0;
var messenger = new EventEmitter();
// Listener for EventEmitter
messenger.on("message", function(msg) {
    console.log(++pos + " MESSAGE: " + msg);
});
// (A) FIRST
console.log(++pos + " FIRST");
//  (B) NEXT
process.nextTick(function() {
  console.log(++pos + " NEXT")
})
// (C) QUICK TIMER
setTimeout(function() {
  console.log(++pos + " QUICK TIMER")
}, 0)
// (D) LONG TIMER
setTimeout(function() {
  console.log(++pos + " LONG TIMER")
}, 10)
// (E) IMMEDIATE
setImmediate(function() {
  console.log(++pos + " IMMEDIATE")
})
// (F) MESSAGE HELLO!
messenger.emit("message", "Hello!");
// (G) FIRST STAT
fs.stat(__filename, function() {
    console.log(++pos + " FIRST STAT");
});
// (H) LAST STAT
fs.stat(__filename, function() {
    console.log(++pos + " LAST STAT");
});
// (I) LAST
console.log(++pos + " LAST");
```

The output of is program is:

1. **FIRST (A).**
2. **MESSAGE: Hello! (F).**
3. **LAST (I).**

4. **NEXT (B).**
5. **QUICK TIMER (C).**
6. **FIRST STAT (G).**
7. **LAST STAT (H).**
8. **IMMEDIATE (E).**
9. **LONG TIMER (D).**

Let's break the preceding code down:

A, F, and I execute in the main program flow and as such they will have the first priority in the main thread (this is obvious; your JavaScript executes its instructions in the order they are written, including the synchronous execution of the `emit` callback).

With the main call stack exhausted, the event loop is now almost reading to process I/O operations. This is the moment when nextTick requests are honored slotting in at the head of the event queue. This is when B is displayed.

The rest of the order should be clear. Timers and I/O operations will be processed next, (C, G, H) followed by the results of the `setImmediate` callback (E), always arriving after any I/O and timer responses are executed.

Finally, the long timeout (D) arrives, being a relatively far-future event.

Notice that re-ordering the expressions in this program will *not* change the output order (outside of possible re-ordering of the STAT results, which only implies that they have been returned from the thread pool in different order, remaining as a group in the correct order as relates to the event queue).

Callbacks and errors

Members of the Node community develop new packages and projects every day. Because of Node's evented nature, callbacks permeate these codebases. We've considered several of the key ways in which events might be queued, dispatched, and handled through the use of callbacks. Let's spend a little time outlining the best practices, in particular about conventions for designing callbacks and handling errors, and discuss some patterns useful when designing complex chains of events and callbacks.

Conventions

Luckily, Node creators agreed upon sane conventions on how to structure callbacks early on. It is important to follow this tradition. Deviation leads to surprises, sometimes very bad surprises, and in general to do so automatically makes an API awkward, a characteristic other developers will rapidly tire of.

One is either returning a function result by executing a callback, handling the arguments received by a callback, or designing the signature for a callback within your API. Whichever situation is being considered, one should follow the convention relevant to that case:

- The first argument returned to a callback function is any error message, preferably in the form of an error object. If no error is to be reported, this slot should contain a `null` value.
- When passing a callback to a function it should be assigned the last slot of the function signature. APIs should be consistently designed this way.
- Any number of arguments may exist between the error and the callback slots.

To create an error object:

```
new Error("Argument must be a String!")
```

Know your errors

It is excellent that the Node community has automatically adopted a convention that compels developers to be diligent and report errors. However, what does one do with errors once they are received?

It is generally a very good idea to centralize error handling in a program. Often, a custom error handling system will be designed, which may send messages to clients, add to a log, and so on. Sometimes it is best to `throw` errors, halting the process.

Node provides more advanced tools for error handling. In particular, Node's `domain` system helps with a problem that evented systems have: how can a stack trace be generated if the full route of a call has been obliterated as it jumped from callback to callback?

The goal of `domain` is simple: fence and label an execution context such that all events that occur within it are identified as such, allowing more informative stack traces. By creating several different domains for each significant segment of your program, a chain of errors can be properly understood.

Additionally, this provides a way to catch errors and handle them, rather than allowing your entire Node process to collapse.

In the following example we're going to create two domains: `appDomain` and `fsDomain`. The goal is to be able to trace which part of our application is in an error state:

```
var domain = require("domain");
var fs = require("fs");

var fsDomain = domain.create();
fsDomain.on("error", function(err) {
  console.error("FS error", err);
});

var appDomain = domain.create();
appDomain.on('error', function(err) {
    console.log("APP error", err);
});
```

We now wrap the main program in `appDomain`, and the filesystem calls in `fsDomain`. We then create an error in `fsDomain` by trying to open a non-existent file:

```
appDomain.run(function() {
    process.nextTick(function() {
        fsDomain.run(function() {
            fs.open('no_file_here', 'r', function(err, fd) {
                if(err) {
                    throw err;
                }
                appDomain.dispose();
            });
        });
    });
});
```

When the preceding code executes, something resembling this should be echoed to the terminal:

```
FS error { [Error: ENOENT, open 'non-existent file']
  errno: 34,
  code: 'ENOENT',
  path: 'non-existent file',
  domain:
   { domain: null,
     _events: { error: [Function] },
     _maxListeners: 10,
     members: [] },
  domainThrown: true }
```

Now let's create an error in `appDomain` by adding this code, which will produce a reference error (as no `b` is defined):

```
appDomain.run(function() {
    a = b;
    process.nextTick(function() {
...
```

An error similar to that in the precious code should be generated and reported by `appDomain`.

Notice the command `appDomain.dispose`. As maintaining these error contexts will consume some memory, it is best to dispose of them when no longer needed—after the code they contain has successfully executed, for example. We'll learn more advanced uses of this tool as we progress into more complex territories.

As an application grows in complexity it will become more and more useful to be able to trap errors and handle them properly, perhaps restarting only one part of an application when it fails rather than the entire system.

Building pyramids

Simplifying control flows has been a concern of the Node community since the very beginning of the project. Indeed, this potential criticism was one of the very first anticipated by Ryan Dahl, who discussed it at length during the talk in which he introduced Node to the JavaScript developer community.

Because deferred code execution often requires the nesting of callbacks within callbacks a Node program can sometimes begin to resemble a sideways pyramid, also known as "The Pyramid of Doom".

Accordingly, there are several Node packages available which take the problem on, employing strategies as varied as futures, fibers, even C++ modules exposing system threads directly. The reader is encouraged to experiment with these:

Async	`https://github.com/caolan/async`
Tame	`https://github.com/maxtaco/tamejs`
Fibers	`https://github.com/laverdet/node-fibers`
Promises	`https://github.com/kriskowal/q`

A more interesting general point is available here for us to consider regarding API choices in Node. Dahl might have reacted to this criticism by, for example, making one of the listed libraries part of Node's core, or indeed changing the entire way JavaScript is written. Instead, it was left to the community to determine the best practices, and to write the relevant packages. This is the *Node* way.

Mikeal Rogers, in discussing why Promises were removed from the Node core, makes a strong argument in the following link for why leaving feature development to the community leads to a stronger core product:

`http://www.futurealoof.com/posts/broken-promises.html`

Considerations

Any developer is regularly making decisions with a far-reaching impact. It is very hard to predict all the possible consequences resulting from a new bit of code or a new design theory. For this reason, it may be useful to keep the shape of your code simple, and to force yourself to consistently follow the common practices of other Node developers. These are some guidelines you may find useful, as follows:

- Generally, try to aim for shallow code. This type of refactoring is uncommon in non-evented environments—remind yourself of it by regularly re-evaluating entry and exit points, and shared functions.
- Where possible provide a common context for callback re-entry. Closures are very powerful tools in JavaScript, and by extension, Node. As long as the context frame length of the enclosed callbacks is not excessive.
- Name you functions. In addition to being useful in deeply recursive constructs, debugging code is much easier when a stack trace contains distinct function names, as opposed to `anonymous`.
- Think hard about priorities. Does the order, in which a given result arrives or a callback is executed, actually matter? Importantly, does it matter in relation to I/O operations? If so, consider `nextTick` and `setImmediate`.
- Consider using finite state machines for managing your events. State machines are (surprisingly) under-represented in JavaScript codebases. When a callback re-enters program flow it has likely changed the state of your application, and the issuing of the asynchronous call itself is a likely indicator that state is about to change.

Listening for file changes

Let's apply what we've learned. The goal is to create a server that a client can connect to and receive updates from Twitter. We will first create a process to query Twitter for any messages with the **hashtag** #nodejs, and writes any found messages to a `tweets.txt` file in 140-byte chunks. We will then create a network server that broadcasts these messages to a single client. Those broadcasts will be triggered by write events on the `tweets.txt` file. Whenever a write occurs, 140-byte chunks are asynchronously read from the last known client read pointer. This will happen until we reach the end of the file, broadcasting as we go. Finally, we will create a simple `client.html` page, which asks for, receives, and displays these messages.

While this example is certainly contrived, it demonstrates:

- Listening to the filesystem for changes and responding to those events
- Using data stream events for reading and writing files
- Responding to network events
- Using timeouts for polling state
- Using a Node server itself as a network event broadcaster

To handle server broadcasting we are going to use the **Server Sent Events** (**SSE**) protocol, a new protocol being standardized as part of HTML5.

We're first going to create a Node server that listens for changes on a file and broadcasts any new content to the client. Open your editor and create a file `server.js`:

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

var theUser = null;
var userPos = 0;
var tweetFile = "tweets.txt";
```

We will be accepting a single user connection, whose pointer will be `theUser`. The `userPos` will store the last position this client read from in `tweetFile`:

```
http.createServer(function(request, response) {
    response.writeHead(200, {
      'Content-Type': 'text/event-stream',
      'Cache-Control': 'no-cache',
      'Access-Control-Allow-Origin': '*'
    });

    theUser = response;

    response.write(':' + Array(2049).join(' ') + '\n');
```

```
        response.write('retry: 2000\n');

        response.socket.on('close', function() {
          theUser = null;
        });

    }).listen(8080);
```

Create an HTTP server listening on port 8080, which will listen for and handle a single connection, storing the `response` argument, representing the pipe connecting the server to client. The `response` argument implements the writeable stream interface, allowing us to write messages to the client:

```
var sendNext = function(fd) {
    var buffer = new Buffer(140);
    fs.read(fd, buffer, 0, 140, userPos * 140, function(err, num) {
        if(!err && num > 0 && theUser) {
            ++userPos;
            theUser.write('data: ' + buffer.toString('utf-8', 0,
              num) + '\n\n');
            return process.nextTick(function() {
                sendNext(fd);
            });
        }
    });
};
```

We create a function to send the client messages. We will be pulling buffers of 140 bytes out of the readable stream bound to our `tweets.txt` file, incrementing our file position counter by one on each read. We write this buffer to the writeable stream binding our server to the client. When done, we queue up a repeat call of the same function using `nextTick`, repeating until we get an error, receive no data, or the client disconnects:

```
function start() {
    fs.open(tweetFile, 'r', function(err, fd) {
        if(err) {
            return setTimeout(start, 1000);
        }
        fs.watch(tweetFile, function(event, filename) {
            if(event === "change") {
               sendNext(fd);
            }
        });
    });
};
start();
```

Finally, we start the process by opening the `tweets.txt` file and watch for any changes, calling `sendNext` whenever new tweets are written. When we start the server there may not yet exist a file to read from, so we poll using `setTimeout` until one exists.

Now that we have a server looking for file changes to broadcast, we need to generate data. We first install the **TWiT** Twitter package for Node, via **npm**.

We then create a process whose sole job is to write new data to a file:

```
var fs    = require("fs");
var Twit = require('twit');

var twit = new Twit({
    consumer_key:         'your key',
      consumer_secret:      'your secret',
      access_token:         'your token',
      access_token_secret:  'your secret token'
})
```

To use this example, you will need a Twitter developer account. Alternatively, there is also the option of changing the relevant code, in the following, to simply write random 140-byte strings to `tweets.txt`.

```
var tweetFile = "tweets.txt";
var writeStream = fs.createWriteStream(tweetFile, {
    flags    : "a"
});
```

This establishes a stream pointer to the same file that our server will be watching. We will be writing to this file:

```
var cleanBuffer = function(len) {
    var buf = new Buffer(len);
    buf.fill('\0');
    return buf;
}
```

Because Twitter messages are never longer than 140 bytes we can simplify the read/write operation by always writing 140-byte chunks, even if some of that space is empty. Once we receive updates we will create a buffer that is *number of messages* x *140* bytes wide, and write those 140-byte chunks to this buffer:

```
var check = function() {

    twit.get('search/tweets', {
        q: '#nodejs since:2013-01-01'
    }, function(err, reply) {
        var buffer = cleanBuffer(reply.statuses.length * 140);
        reply.statuses.forEach(function(obj, idx) {
            buffer.write(obj.text, idx*140, 140);
        });
        writeStream.write(buffer);
    })
    setTimeout(check, 10000);
};
check();
```

We now create a function that will be asked every ten seconds to check for messages containing the hashtag `#nodejs`. Twitter returns an array of message objects. The one object property we are interested in is the `#text` of the message. Calculate the number of bytes necessary to represent these new messages (*140* x *message count*), fetch a clean buffer, and fill it with 140-byte chunks until all messages are written. Finally, this data is written to our `tweets.txt` file, causing a change event to occur that our server is notified of.

The final piece is the client page itself. This is a rather simple page, and how it operates should be familiar to the reader. The only thing to note is the use of SSE that listens to port 8080 on localhost. It should be clear how, on receipt of a new tweet from the server, a list element is added to the unordered list container `#list`:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>

<script>

window.onload = function() {
    var list      = document.getElementById("list");
    var evtSource = new
      EventSource("http://localhost:8080/events");
```

```
    evtSource.onmessage = function(e) {
      var newElement = document.createElement("li");
      newElement.innerHTML = e.data;
      list.appendChild(newElement);
    }
}

</script>

<body>

<ul id="list"></ul>

</body>
</html>
```

To read more about SSE refer *Chapter 6*, *Creating Real-time Applications*, or you can visit the following link:

`https://developer.mozilla.org/en-US/docs/Server-sent_events/Using_server-sent_events`

Summary

Programming with events is not always easy. The control and context switches, defining the paradigm often confound those new to evented systems. This seemingly reckless loss of control and the resulting complexity drives many developers away from these ideas. Students in introductory programming courses normally develop a mindset in which program flow can be dictated, where a program whose execution flow does not proceed sequentially from A to B can bend understanding.

By examining the evolution of the architectural problems Node is now attempting to solve for network applications—in terms of scaling, in terms of code organization, in general terms of data and complexity volume, in terms of state awareness, and in terms of well-defined data and process boundaries—we have learned how managing these event queues can be done intelligently. We have seen how different event sources are predictably stacked for an event loop to process, and how far-future events can enter and re-enter contexts using closures and smart callback ordering.

We now have a basic domain understanding of the design and characteristics of Node, in particular how evented programming is done using it. Let's now move into larger, more advanced applications of this knowledge.

3
Streaming Data Across Nodes and Clients

A jug fills drop by drop.

— *Buddha*

We now have a clearer picture of how the evented, I/O-focused design ethic of Node is reflected across its various module APIs, delivering a consistent and predictable environment for development. In this chapter we will discover how data, of many shapes and sizes, pulled from files or other sources, can be read, written, and manipulated just as easily using Node. Ultimately we will learn how to use Node to develop networked servers with rapid I/O interfaces that support highly concurrent applications sharing real-time data across thousands of clients simultaneously.

Those who work with Internet-based software will have heard about "Big Data". The importance of I/O efficiency is not lost on those witnessing this explosive growth in data volume.

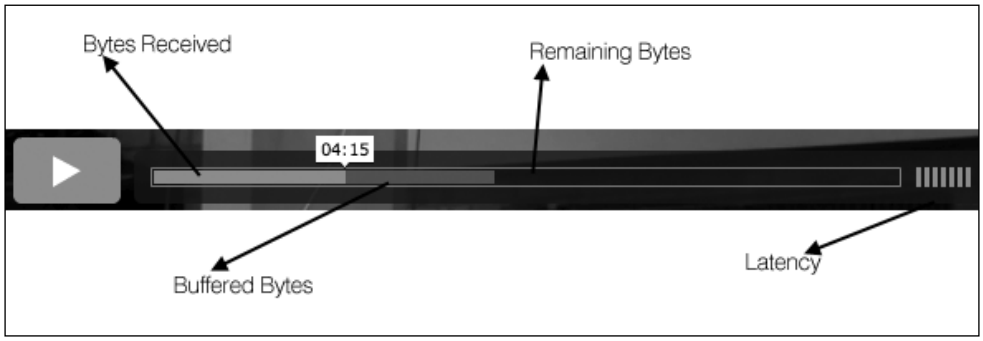
Implied by this expansion in data is an increase in the size of individual media objects transmitted over the network. The videos being watched are longer. The images being viewed are bigger. The searches are wider and the result sets are larger.

Additionally, scaled network applications are normally spread across many servers, requiring that the processing of data streams be distributed across processes. Additionally, the funneling of disparate streams into a single custom output stream has become the signature of cloud services and various other APIs.

Anyone who has visited a website specializing in the display of large media objects (such as videos) has likely also experienced latency. In an ideal world, one would make a request to a server for `lolcats.mpg` and would receive this file instantly.

However, moving large amounts of data (bytes) takes time. As the size of files accessible over the Internet began to grow in size, from some number of kilobytes to many gigabytes, download times grew in parallel, from several minutes to several hours. Originally, in order to view a video, one's media viewer (often a browser) required a complete copy prior to commencing playback. For the user, this meant long waiting times spent staring at the progress bar or spinning icons. It took very little vision to see that this solution was untenable.

Because the demand was so high for large media objects, those who had an interest in distributing these objects encouraged the development of better distribution mechanisms. As Internet became the world's de-facto information distribution network, for both fixed media objects and live data broadcasts, better solutions arrived. In particular, *streaming* media became the standard. The reader will certainly be familiar with streaming media, and in particular the idea of *buffering* a stream. Small chunks (for example a video stream is "played" as they are received) arriving too early are buffered (helping with "hiccups" and latency), until the entire file has arrived:



Here, a streaming file is simply a stream of data partitioned into slices, where each slice can be viewed independently irrespective of the availability of others. One can write to a data stream, or listen on a data stream, free to dynamically allocate bytes, to ignore bytes, to re-route bytes. Streams of data can be chunked, many processes can share chunk handling, chunks can be transformed and reinserted, and data flows can be precisely emitted and creatively managed.

Recalling our discussion on modern software and the **Rule of Modularity**, we can see how streams facilitate the creation of independent share-nothing processes that do one task well, and in combination can compose a predictable architecture whose complexity does not preclude an accurate appraisal of its shape. If the interfaces to data are uncontroversial, the data map can be accurately modeled independent of considerations about data volume or routing.

Managing I/O in Node involves managing data events bound to data streams. A Node `Stream` object is an instance of `EventEmitter`. This abstract interface is implemented in various Node modules and objects. Let's begin by understanding Node's `Stream` module, then move on to a discussion of how network I/O in Node is handled via various `Stream` implementations; in particular, the HTTP module.

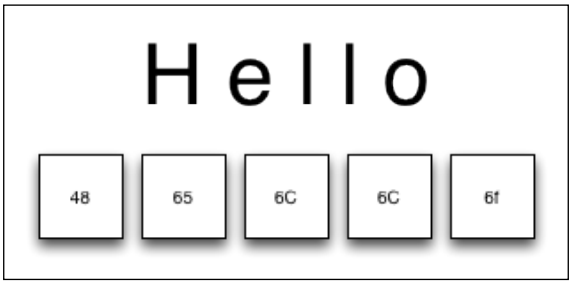
Exploring streams

According to *Bjarne Stoustrup* in his book *The C++ Programming Language, Third Edition*:

> *Designing and implementing a general input/output facility for a programming language is notoriously difficult... An I/O facility should be easy, convenient, and safe to use; efficient and flexible; and, above all, complete.*

It shouldn't surprise anyone that a design team, focused on providing efficient and easy I/O, has delivered such a facility through Node. Through a symmetrical and simple interface, which handles data buffers and stream events so that the implementer does not have to, Node's `Stream` module is the preferred way to manage asynchronous data streams for both internal modules and, hopefully, for the modules developers will create.

A stream in Node is simply a sequence of bytes. At any time, a stream contains a buffer of bytes, and this buffer has a zero or greater length:



Because each character in a stream is well defined, and because every type of digital data can be expressed in bytes, any part of a stream can be redirected, or "piped", to any other stream, different chunks of the stream can be sent do different handlers, and so on. In this way stream input and output interfaces are both flexible and predictable and can be easily coupled.

Digital streams are well described using the analogy of fluids, where individual bytes (drops of water) are being pushed through a pipe. In Node, streams are objects representing data flows that can be written to and read from asynchronously. The Node philosophy is a non-blocking flow, I/O is handled via streams, and so the design of the `Stream` API naturally duplicates this general philosophy. In fact, there is no other way of interacting with streams except in an asynchronous, evented manner—you are prevented, by design, from blocking I/O.

Five distinct base classes are exposed via the abstract `Stream` interface: `Readable`, `Writable`, `Duplex`, `Transform`, and `PassThrough`. Each base class inherits from `EventEmitter`, which we know of as an interface to which event listeners and emitters can be bound.

As we will learn, and here will emphasize, the `Stream` interface is an **abstract** interface. An abstract interface functions as a kind of blueprint or definition, describing the features that must be built into each constructed instance of a `Stream` object. For example, a `Readable` stream implementation is required to implement a public read method which delegates to the interface's internal `_read` method.

In general, all stream implementations should follow these guidelines:

- As long as data exists to send, `write` to a stream until that operation returns `false`, at which point the implementation should wait for a `drain` event, indicating that the buffered stream data has emptied
- Continue to call `read` until a null value is received, at which point wait for a `readable` event prior to resuming reads
- Several Node I/O modules are implemented as streams. Network sockets, file readers and writers, `stdin` and `stdout`, `zlib`, and so on. Similarly, when implementing a readable data source, or data reader, one should implement that interface as a `Stream` interface.

It is important to note that as of Node 0.10.0 the `Stream` interface changed in some fundamental ways. The Node team has done its best to implement backwards-compatible interfaces, such that (most) older programs will continue to function without modification. In this chapter we will not spend any time discussing the specific features of this older API, focusing on the current (and future) design. The reader is encouraged to consult Node's online documentation for information on migrating older programs.

Implementing readable streams

Streams producing data that another process may have an interest in are normally implemented using a `Readable` stream. A `Readable` stream saves the implementer all the work of managing the read queue, handling the emitting of data events, and so on.

To create a `Readable` stream:

```
var stream = require('stream');
var readable = new stream.Readable({
    encoding : "utf8",
    highWaterMark : 16000,
    objectMode: true
});
```

As previously mentioned, `Readable` is exposed as a base class, which can be initialized through three options:

- **encoding**: Decode buffers into the specified encoding, defaulting to UTF-8.
- **highWaterMark**: Number of bytes to keep in the internal buffer before ceasing to read from the data source. The default is 16 KB.
- **objectMode**: Tell the stream to behave as a stream of objects instead of a stream of bytes, such as a stream of JSON objects instead of the bytes in a file. Default `false`.

In the following example we create a mock `Feed` object whose instances will inherit the `Readable` stream interface. Our implementation need only implement the abstract `_read` method of `Readable`, which will `push` data to a consumer until there is nothing more to push, at which point it triggers the `Readable` stream to emit an "end" event by pushing a `null` value:

```
var Feed = function(channel) {
    var readable = new stream.Readable({
        encoding : "utf8"
    });
    var news = [
        "Big Win!",
        "Stocks Down!",
        "Actor Sad!"
    ];
    readable._read = function() {
        if(news.length) {
            return readable.push(news.shift() + "\n");
        }
        readable.push(null);
    };
    return readable;
}
```

Now that we have an implementation, a consumer might want to instantiate the stream and listen for stream events. Two key events are **readable** and **end**.

The readable event is emitted as long as data is being pushed to the stream. It alerts the consumer to check for new data via the `read` method of `Readable`.

Note again how the `Readable` implementation must provide a private `_read` method, which services the public `read` method exposed to the consumer API.

The end event will be emitted whenever a `null` value is passed to the `push` method of our `Readable` implementation.

Here we see a consumer using these methods to display new stream data, providing a notification when the stream has stopped sending data:

```
var feed = new Feed();
feed.on("readable", function() {
    var data = feed.read();
    data && process.stdout.write(data);
});
feed.on("end", function() {
    console.log("No more news");
});
```

Similarly, we could implement a stream of objects through the use of the `objectMode` option:

```
    var readable = new stream.Readable({
        objectMode : true
    });
    var prices = [
        { price : 1 },
        { price : 2 }
    ];
...
readable.push(prices.shift());

// > { prices : 1 }
// > { prices : 2 }
```

Here we see that each read event is receiving an object, rather than a buffer or string.

Finally, the `read` method of a `Readable` stream can be passed a single argument indicating the number of bytes to be read from the stream's internal buffer. For example, if it was desired that a file should be read one byte at a time, one might implement a consumer using a routine similar to:

```
readable.push("Sequence of bytes");
...
feed.on("readable", function() {
    var character;
    while(character = feed.read(1)) {
        console.log(character);
    };
});
// > S
// > e
// > q
// > ...
```

Here it should be clear that the `Readable` stream's buffer was filled with a number of bytes all at once, but was read from discretely.

Pushing and pulling

We have seen how a `Readable` implementation will use `push` to populate the stream buffer for reading. When designing these implementations it is important to consider how volume is managed, at either end of the stream. Pushing more data into a stream than can be read can lead to complications around exceeding available space (memory). At the consumer end it is important to maintain awareness of termination events, and how to deal with pauses in the data stream.

One might compare the behavior of data streams running through a network with that of water running through a hose.

As with water through a hose, if a greater volume of data is being pushed into the read stream than can be efficiently drained out of the stream at the consumer end through `read`, a great deal of back pressure builds, causing a data backlog to begin accumulating in the stream object's buffer. Because we are dealing with strict mathematical limitations, `read` simply cannot be compelled to release this pressure by reading more quickly—there may be a hard limit on available memory space, or other limitation. As such, memory usage can grow dangerously high, buffers can overflow, and so forth.

A stream implementation should therefore be aware of, and respond to, the response from a `push` operation. If the operation returns `false` this indicates that the implementation should cease reading from its source (and cease pushing) until the next `_read` request is made.

In conjunction with the above, if there is no more data to push *but more is expected in the future* the implementation should `push` an empty string (`""`), which adds no data to the queue but does ensure a future `readable` event.

While the most common treatment of a stream buffer is to `push` to it (queuing data in a line), there are occasions where one might want to place data on the front of the buffer (jumping the line). Node provides an `unshift` operation for these cases, which behavior is identical to push, outside of the aforementioned difference in buffer placement.

Writable streams

A `Writable` stream is responsible for accepting some value (a stream of bytes, a string) and writing that data to a destination. Streaming data into a file container is a common use case.

To create a `Writable` stream:

```
var stream = require('stream');
var readable = new stream.Writable({
    highWaterMark : 16000,
    decodeStrings: true
});
```

The `Writable` streams constructor can be instantiated with two options:

- **highWaterMark**: The maximum number of bytes the stream's buffer will accept prior to returning false on writes. Default is 16 KB
- **decodeStrings**: Whether to convert strings into buffers before writing. Default is true.

As with `Readable` streams, custom `Writable` stream implementations must implement a `_write` handler, which will be passed the arguments sent to the `write` method of instances.

One should think of a `Writable` stream as a data target, such as for a file you are uploading. Conceptually this is not unlike the implementation of push in a `Readable` stream, where one pushes data until the data source is exhausted, passing `null` to terminate reading. For example, here we write 100 bytes to `stdout`:

```
var stream = require('stream');
var writable = new stream.Writable({
    decodeStrings: false
});
writable._write = function(chunk, encoding, callback) {
    console.log(chunk);
    callback();
}
var w = writable.write(new Buffer(100));
writable.end();
console.log(w); // Will be `true`
```

There are two key things to note here.

First, our `_write` implementation fires the `callback` function immediately after writing, a callback that is always present, regardless of whether the instance `write` method is passed a callback directly. This call is important for indicating the status of the write attempt, whether a failure (error) or a success.

Second, the call to `write` returned `true`. This indicates that the internal buffer of the `Writable` implementation has been emptied after executing the requested write. What if we sent a very large amount of data, enough to exceed the default size of the internal buffer? Modifying the above example, the following would return `false`:

```
var w = writable.write(new Buffer(16384));
console.log(w); // Will be 'false'
```

The reason this write returns false is that it has reached the `highWaterMark` option—default value of 16 KB (*16 * 1024*). If we changed this value to 16383, `write` would again return true (or one could simply increase its value).

What to do when `write` returns false? One should certainly not continue to send data! Returning to our metaphor of water in a hose: when the stream is full, one should wait for it to drain prior to sending more data. Node's `Stream` implementation will emit a `drain` event whenever it is safe to write again. When `write` returns `false` listen for the `drain` event before sending more data.

Putting together what we have learned, let's create a `Writable` stream with a `highWaterMark` value of 10 bytes. We will send a buffer containing more than 10 bytes (composed of `A` characters) to this stream, triggering a `drain` event, at which point we write a single `Z` character. It should be clear from this example that Node's `Stream` implementation is managing the **buffer overflow** of our original payload, warning the original write method of this overflow, performing a controlled depletion of the internal buffer, and notifying us when it is safe to `write` again:

```
var stream = require('stream');
var writable = new stream.Writable({
    highWaterMark: 10
});
writable._write = function(chunk, encoding, callback) {
    process.stdout.write(chunk);
    callback();
}
writable.on("drain", function() {
    writable.write("Z\n");
});
var buf = new Buffer(20, "utf8");
buf.fill("A");

console.log(writable.write(buf.toString())); // false
```

The result should be a string of 20 `A` characters, followed by `false`, then followed by the character `Z`.

The fluid data in a `Readable` stream can be easily redirected to a `Writable` stream. For example, the following code will take any data sent by a terminal (`stdin` is a `Readable` stream) and pass it to the destination `Writable` stream, `stdout`:

```
process.stdin.pipe(process.stdout);
```

Whenever a `Writable` stream is passed to a `Readable` stream's `pipe` method, a **pipe** event will fire. Similarly, when a `Writable` stream is removed as a destination for a `Readable` stream, the **unpipe** event fires.

To remove a `pipe`, use the following:

```
unpipe(destination stream)
```

Duplex streams

A duplex stream is both readable and writeable. For instance, a TCP server created in Node exposes a socket that can be both read from and written to:

```
var stream     = require("stream");
var net      = require("net");
net
.createServer(function(socket) {
    socket.write("Go ahead and type something!");
    socket.on("readable", function() {
        process.stdout.write(this.read())
    });
})
.listen(8080);
```

When executed, this code will create a TCP server that can be connected to via Telnet:

```
telnet 127.0.0.1 8080
```

Upon connection, the connecting terminal will print out **Go ahead and type something!**—*writing* to the socket. Any text entered in the connecting terminal will be echoed to the `stdout` of the terminal running the TCP server (*reading* from the socket). This implementation of a bi-directional (duplex) communication protocol demonstrates clearly how independent processes can form the nodes of a complex and responsive application, whether communicating across a network or within the scope of a single process.

The options sent when constructing a `Duplex` instance merge those sent to `Readable` and `Writable` streams, with no additional parameters. Indeed, this stream type simply assumes both roles, and the rules for interacting with it follow the rules for the interactive mode being used.

As a Duplex stream assumes both read and write roles, any implementation is required to implement both `_write` and `_read` methods, again following the standard implementation details given for the relevant stream type.

Transforming streams

On occasion stream data needs to be processed, often in cases where one is writing some sort of binary protocol or other "on the fly" data transformation. A `Transform` stream is designed for this purpose, functioning as a `Duplex` stream that sits between a `Readable` stream and a `Writable` stream.

A `Transform` stream is initialized using the same options used to initialize a typical `Duplex` stream. Where `Transform` differs from a normal `Duplex` stream is in its requirement that the custom implementation merely provide a `_transform` method, excluding the `_write` and `_read` method requirement.

The `_transform` method will receive three arguments, first the sent buffer, an optional encoding argument, and finally a callback which `_transform` is expected to call when the transformation is complete:

```
_transform = function(buffer, encoding, cb) {
    var transformation = "...";
    this.push(transformation)
    cb();
}
```

Let's imagine a program that wishes to convert **ASCII** (**American Standard Code for Information Interchange**) codes into ASCII characters, receiving input from `stdin`. We would simply pipe our input to a `Transform` stream, then piping its output to `stdout`:

```
var stream = require('stream');
var converter = new stream.Transform();
converter._transform = function(num, encoding, cb) {
    this.push(String.fromCharCode(new Number(num)) + "\n")
    cb();
}
process.stdin.pipe(converter).pipe(process.stdout);
```

Interacting with this program might produce an output resembling the following:

```
65 A
66 B
256 Ā
257 ā
```

An example of a transform stream will be demonstrated in the example that ends this chapter.

Using PassThrough streams

This sort of stream is a trivial implementation of a Transform stream, which simply passes received input bytes through to an output stream. This is useful if one doesn't require any transformation of the input data, and simply wants to easily pipe a `Readable` stream to a `Writable` stream.

`PassThrough` streams have benefits similar to JavaScript's anonymous functions, making it easy to assert minimal functionality without too much fuss. For example, it is not necessary to implement an abstract base class, as one does with for the `_read` method of a `Readable` stream. Consider the following use of a `PassThrough` stream as an event spy:

```
var fs = require('fs');
var stream = new require('stream').PassThrough();
spy.on('end', function() {
    console.log("All data has been sent");
});
fs.createReadStream("./passthrough.js").pipe(spy).pipe(process.std
  out);
```

Creating an HTTP server

HTTP is a stateless data transfer protocol built upon a request/response model: clients make requests to servers, which then return a response. Facilitating this sort of rapid patter network communication is the sort of I/O Node is designed to excel at, it has unsurprisingly become identified as primarily a toolkit for creating servers—though it can certainly be used to do much, much more. Throughout this book we will be creating many implementations of HTTP servers, as well as other protocol servers, and will be discussing best practices in more depth, contextualized within specific business cases. It is expected that you have already had some experience doing the same. For both of these reasons we will quickly move through a general overview into some more specialized uses.

At its simplest, an HTTP server responds to connection attempts, and manages data as it arrives and as it is sent along. A Node server is typically created using the `createServer` method of the http module:

```
var http = require('http');
var server = http.createServer(function(request, response) {
    console.log("Got Request Headers:");
    console.log(request.headers);
    response.writeHead(200, {
        'Content-Type': 'text/plain'
    });
    response.write("PONG");
    response.end();
});

server.listen(8080);
```

The object returned by `http.createServer` is an instance of `http.Server`, which extends `EventEmitter`, broadcasting network events as they occur, such as a client connection or request. The above code is a common way to write Node servers. However, it is worth pointing out that directly instantiating the `http.Server` class is sometimes a useful way to distinguish distinct server/client interactions. We will use that format for the following examples.

Here, we create a basic server that simply reports when a connection is made, and when it is terminated:

```
var http = require('http');
var server = new http.Server();
server.on("connection", function(socket) {
    console.log("Client arrived: " + new Date());
    socket.on("end", function() {
        console.log("Client left: " + new Date());
    });
})
server.listen(8080);
```

When building multiuser systems, especially authenticated multiuser systems, this point in the server-client transaction is an excellent place for client validation and tracking code, including setting or reading of cookies and other session variables, or the broadcasting of a client arrival event to other clients working together in a concurrent real-time application.

By adding a listener for requests we arrive at the more common request/response pattern, handled as a `Readable` stream. When a client POSTs some data, we can catch that data like the following:

```
server.on("request", function(request, response) {
    request.setEncoding("utf8");
    request.on("readable", function() {
        console.log(request.read())
    });
});
```

Try sending some data to this server using **curl**:

```
curl http://localhost:8080 -d "Here is some data"
```

By using connection events we can nicely separate our connection handling code, grouping it into clearly defined functional domains correctly described as executing in response to particular events.

For example, we can set timers on server connections. Here we terminate client connections that fail to send new data within a roughly two second window:

```
server.setTimeout(2000, function(socket) {
    socket.write("Too Slow!", "utf8");
    socket.end();
});
```

If one simply wants to set the number of milliseconds of inactivity before a socket is presumed to have timed out, simply use `server.timeout = (Integer)num_milliseconds`. To disable socket timeouts, pass a value of 0(zero).

Let's now take a look at how Node's HTTP module can be used to enter into more interesting network interactions.

Making HTTP requests

It is often necessary for a network application to make external HTTP calls. HTTP servers are also often called upon to perform HTTP services for clients making requests. Node provides an easy interface for making external HTTP calls.

For example, the following code will fetch the front page of `google.com`:

```
var http = require('http');
http.request({
    host: 'www.google.com',
    method: 'GET',
    path: "/"
}, function(response) {
    response.setEncoding("utf8");
    response.on("readable", function() {
        console.log(response.read())
    });
}).end();
```

As we can see, we are working with a Readable stream, which can be written to a file.

A popular Node module for managing HTTP requests is *Mikeal Rogers*' **request**:

`https://github.com/mikeal/request`

Because it is common to use `HTTP.request` in order to GET external pages, Node offers a shortcut:

```
http.get("http://www.google.com/", function(response) {
  console.log("Status: " + response.statusCode);
}).on('error', function(err) {
  console.log("Error: " + err.message);
});
```

Let's now look at some more advanced implementations of HTTP servers, where we perform general network services for clients.

Proxying and tunneling

Sometimes it is useful to provide a means for one server to function as a proxy, or broker, for other servers. This would allow one server to distribute load to other servers, for example. Another use would be to provide access to a secured server to users who are unable to connect to that server directly. It is also common to have one server answering for more than one URL—by using a proxy, that one server can forward requests to the right recipient.

Because Node has a consistent streams interface throughout its network interfaces, we can build a simple HTTP proxy in just a few lines of code. For example, the following program will set up an HTTP server on port 8080 which will respond to any request by fetching the front page of Google and piping that back to the client:

```
var http = require('http');
var server = new http.Server();
server.on("request", function(request, socket) {
    http.request({
        host: 'www.google.com',
        method: 'GET',
        path: "/",
        port: 80
    }, function(response) {
        response.pipe(socket);
    }).end();
});
server.listen(8080);
```

Once this server receives the client socket, it is free to push content from any readable stream back to the client, and here the result of GET of `www.google.com` is so streamed. One can easily see how an external content server managing a caching layer for your application might become a proxy endpoint, for example.

Using similar ideas we can create a tunneling service, using Node's native CONNECT support. Tunneling involves using a proxy server as an intermediary to communicate with a remote server on behalf of a client. Once our proxy server connects to a remote server, it is able to pass messages back and forth between that server and a client. This is advantageous when a direct connection between a client and a remote server is not possible, or not desired.

First, we'll set up a proxy server responding to HTTP CONNECT requests, then make a CONNECT request to that server. The proxy receives our client's `Request` object, the client's socket itself, and the **head** (the first packet) of the tunneling stream. We then open the requested remote network socket. All that is left to do is creating the tunnel, which we do by piping remote data to the client, and client data to the remote connection:

```
var http = require('http');
var net = require('net');
var url = require('url');
var proxy = new http.Server();
proxy.on('connect', function(request, clientSocket, head) {
    var reqData = url.parse('http://' + request.url);
      var remoteSocket = net.connect(reqData.port, reqData.hostname,
function() {
        clientSocket.write('HTTP/1.1 200 \r\n\r\n');
        remoteSocket.write(head);
        remoteSocket.pipe(clientSocket);
        clientSocket.pipe(remoteSocket);
      });
}).listen(8080);

var request = http.request({
    port: 8080,
    hostname: 'localhost',
    method: 'CONNECT',
    path: 'www.google.com:80'
});
request.end();
request.on('connect', function(res, socket, head) {
    socket.setEncoding("utf8");
    socket.write('GET / HTTP/1.1\r\nHost: www.google.com:80\r\
nConnection: close\r\n\r\n');
    socket.on('readable', function() {
        console.log(socket.read());
    });
    socket.on('end', function() {
        proxy.close();
    });
});
```

HTTPS, TLS (SSL), and securing your server

The security of web applications has become a significant discussion topic in recent years. Traditional applications normally benefited from the well-tested and mature security models designed into the major servers and application stacks underpinning major deployments. For one reason or another, web applications were allowed to venture into the experimental world of client-side business logic and open web services shielded by a diaphanous curtain.

As Node is regularly deployed as a web server, it is imperative that the community begins to accept responsibility for securing these servers. HTTPS is a secure transmission protocol—essentially encrypted HTTP formed by layering the HTTP protocol on top of the SSL/TLS protocol.

Creating a self-signed certificate for development

In order to support SSL connections a server will need a properly signed certificate. While developing, it is much easier to simply create a self-signed certificate, which will allow one to use Node's HTTPS module.

These are the steps needed to create a certificate for development. Note that this process does not create a real certificate and is *not* secure—it simply allows us to develop within a HTTPS environment. From a terminal:

```
openssl genrsa -out server-key.pem 2048
openssl req -new -key server-key.pem -out server-csr.pem
openssl x509 -req -in server-csr.pem -signkey server-key.pem -out
server-cert.pem
```

These keys may now be used to develop HTTPS servers. The contents of these files need simply be passed along as options to a Node server:

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

https.createServer({
    key: fs.readFileSync('server-key.pem'),
    cert: fs.readFileSync('server-cert.pem')
}, function(req,res) {
      ...
}).listen(443)
```

Free *low-assurance* SSL certificates are available from `http://www.startssl.com/` for cases where self-signed certificates are not ideal during development.

Installing a real SSL certificate

In order to move a secure application out of a development environment and into an Internet-exposed environment a real certificate will need to be purchased. The prices of these certificates has been dropping year by year, and it should be easy to find reasonably priced providers of certificates with a high-enough level of security. Some providers even offer free person-use certificates.

Setting up a professional cert simply requires changing the HTTPS options we introduced above. Different providers will have different processes and filenames. Typically you will need to download or otherwise receive from your provider a private `.key` file, your signed domain certificate `.crt` file, and a bundle describing certificate chains:

```
var options = {
    key         : fs.readFileSync("mysite.key"),
    cert    : fs.readFileSync("mysite.com.crt"),
    ca        : [ fs.readFileSync("gd_bundle.crt") ]
};
```

It is important to note that the `ca` parameter must be sent as an *array*, even if the bundle of certificates has been concatenated into one file.

The request object

HTTP request and response messages are similar, consisting of:

- A status line, which for a request would resemble `GET/index.html HTTP/1.1`, and for a response would resemble `HTTP/1.1 200 OK`
- Zero or more headers, which in a request might include `Accept-Charset: UTF-8 or From: user@server.com`, and in responses might resemble `Content-Type: text/html and Content-Length: 1024`
- A message body, which for a response might be an HTML page, and for a POST request might be some form data

We've seen how HTTP server interfaces in Node are expected to expose a request handler, and how this handler will be passed some form of a request and response object, each of which implement a readable or writable stream.

We will cover the handling of POST data and Header data in more depth later in this chapter. Before we do, let's go over how to parse out some of the more straightforward information contained in a request.

The URL module

Whenever a request is made to an HTTP server the request object will contain url property, identifying the targeted resource. This is accessible via `request.url`. Node's URL module is used to decompose a typical URL string into its constituent parts. Consider the following figure:

```
> console.log(url.parse("http://username:password@www.example.org:8080/events/today/?filter=sports&maxresults=20#football"));
{ protocol: 'http:',
  slashes: true,
  auth: 'username:password',
  host: 'www.example.org:8080',
  port: '8080',
  hostname: 'www.example.org',
  hash: '#football',
  search: '?filter=sports&maxresults=20',
  query: 'filter=sports&maxresults=20',
  pathname: '/events/today/',
  path: '/events/today/?filter=sports&maxresults=20',
  href: 'http://username:password@www.example.org:8080/events/today/?filter=sports&maxresults=20#football' }
```

We see how the `url.parse` method decomposes strings, and the meaning of each segment should be clear. It might also be clear that the `query` field would be more useful if it was itself parsed into Key/Value pairs. This is accomplished by passing `true` as the second argument of to the `parse` method, which would change the query field value given above into a more useful key/value map:

```
query: { filter: 'sports', maxresults: '20' }
```

This is especially useful when parsing GET requests.

There is one final argument for `url.parse` that relates to the difference between these two URLs:

- `http://www.example.org`
- `//www.example.org`

The second URL here is an example of a (relatively unknown) design feature of the HTTP protocol: the protocol-relative URL (technically, a **network-path reference**), as opposed to the more common *absolute* URL.

To learn more about how network-path references are used to smooth resource protocol resolution visit the following link:

`http://tools.ietf.org/html/rfc3986#section-4.2`

The issue under discussion is this: `url.parse` will treat a string beginning with slashes as indicating a path, not a host. For example, `url.parse("//www.example.org")` will set the following values in the `host` and `path` fields:

host: null,

path: '//www.example.org'

What we actually want is the reverse:

host: 'www.example.org',

path: null

To resolve this issue, pass `true` as the third argument to `url.parse`, which indicates to the method that slashes denote a host, not a path:

```
url.parse("//www.example.org",null,true)
```

It is also the case that a developer will want to create a URL, such as when making requests via `http.request`. The segments of said URL may be spread across various data structures and variables, and will need to be assembled. One accomplishes this by passing an object like the one returned from `url.parse` to the method `url.format`.

The following code will create the URL string `http://www.example.org`:

```
url.format({
    protocol: 'http:',
      host: 'www.example.org'
})
```

Similarly, one may also use the `url.resolve` method to generate URL strings in the common scenario of requiring the concatenating a base URL and a path:

```
url.resolve("http://example.org/a/b", "c/d") //
'http://example.org/a/c/d'
url.resolve("http://example.org/a/b", "/c/d") //
'http://example.org/c/d'
url.resolve("http://example.org", "http://google.com") //
'http://google.com/'
```

The Querystring module

As we saw with the `URL` module, query strings often need to be parsed into a map of key/value pairs. The `Querystring` module will either decompose an existing query string into its parts, or assemble a query string from a map of key/value pairs.

For example, `querystring.parse("foo=bar&bingo=bango")` will return:

```
{     foo: 'bar',
    bingo: 'bango' }
```

If our query strings are not formatted using the normal "&" separator and "=" assignment character, the `Querystring` module offers customizable parsing. The second argument to Querystring can be a custom separator string, and the third a custom assignment string. For example, the following will return the same mapping as given previously on a query string with custom formatting:

```
var qs = require("querystring");
console.log(qs.parse("foo:bar^bingo:bango", "^", ":"))
// { foo: 'bar', bingo: 'bango' }
```

One can compose a query string using the `Querystring.stringify` method:

```
console.log(qs.stringify({ foo: 'bar', bingo: 'bango' }))
// foo=bar&bingo=bango
```

As with `parse`, `stringify` also accepts custom separator and assignment arguments:

```
console.log(qs.stringify({ foo: 'bar', bingo: 'bango' }, "^",
  ":"))
// foo:bar^bingo:bango
```

Query strings are commonly associated with GET requests, seen following the `?` character. As we've seen above, in these cases automatic parsing of these strings using the url module is the most straightforward solution. However, strings formatted in such a manner also show up when we're handling POST data, and in these cases the `Querystring` module is of real use. We'll discuss this usage shortly. But first, something about HTTP headers.

Working with headers

Each HTTP request made to a Node server will likely contain useful header information, and clients normally expect to receive similar package information from a server. Node provides straightforward interfaces for reading and writing headers. We'll briefly go over those simple interfaces, clarifying some details. Finally, we'll discuss how to more advanced header usage might be implemented in Node, studying some common network responsibilities a Node server will likely need to accommodate.

A typical request header will look something like the following:

```
{ host: '127.0.0.1:8080',
  'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:19.0)
  Gecko/20100101 Firefox/19.0',
  accept:
  'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
  'accept-language': 'en-US,en;q=0.5',
  'accept-encoding': 'gzip, deflate',
  connection: 'keep-alive',
  'if-modified-since': 'Sun Apr 07 2013 08:09:12 GMT-0400 (EDT)',
  'if-none-match': '1441a7909c087dbbe7ce59881b9df8b9',
  'cache-control': 'max-age=0' }
```

Headers are simple Key/Value pairs. Request keys are always lowercased. You may use any case format when setting response keys.

Reading headers is straightforward. Read header information by examining the `request.header` object, which is a 1:1 mapping of the header's Key/Value pairs. To fetch the "accept" header from the previous example, simply read `request.headers.accept`.

The number of incoming headers can be limited by setting the `maxHeadersCount` property of your HTTP server.

If it is preferred that headers are read programmatically, Node provides the `response.getHeader` method, accepting the header key as its first argument.

While request headers are simple Key/Value pairs, when writing headers we need a more expressive interface. As a response typically must send a status code, Node provides a straightforward way to prepare a response status line and header group in one command:

```
response.writeHead(200, {
  'Content-Length': 4096,
  'Content-Type': 'text/plain'
});
```

To set headers individually, one can use `response.setHeader`, passing two arguments: the header key, followed by the header value.

To set multiple headers with the same name, one may pass an array to `response.setHeader`:

```
response.setHeader("Set-Cookie", ["session:12345",
"language=en"]);
```

Occasionally it may be necessary to remove a response header after that header has been "queued". This is accomplished by using `response.removeHeader`, passing the header name to be removed as an argument.

Headers must be written prior to writing a response. It is an error to write a header after a response has been sent.

Using cookies

The HTTP protocol is stateless. Any given request has no information on previous requests. For a server this meant that determining if two requests originated from the same browser was not possible. Cookies were invented to solve this problem. Cookies are primarily used to share state between clients (usually a browser) and a server, existing as small text files stored in browsers.

Cookies are insecure. Cookie information flows between a server and a client in plain text. There is any number of tamper points in between. Browsers allow easy access to them, for example. This is a good idea, as nobody wants information on their browser or local machine to be hidden from them, beyond their control.

Nevertheless, cookies are also used rather extensively to maintain state information, or pointers to state information, particularly in the case of user sessions or other authentication scenarios.

It is assumed that you are familiar with how cookies function in general. Here we will discuss how cookies are fetched, parsed, and set by a Node HTTP server. We will use the example of a server that echoes back the value of a sent cookie. If no cookie exists, the server will create that cookie and instruct the client to ask for it again.

First we create a server that checks request headers for cookies:

```
var http = require('http');
var url = require('url');
var server = http.createServer(function(request, response) {
    var cookies = request.headers.cookie;
```

Note that cookies are stored as the `cookie` attribute of `request.headers`. If no cookies exist for this domain, we will need to create one, giving it the name `session` and a value of `123456`:

```
    if(!cookies) {
        var cookieName = "session";
        var cookieValue = "123456";
        var expiryDate = new Date();
        expiryDate.setDate(expiryDate.getDate() + 1);
        var cookieText = cookieName + '=' + cookieValue + ';expires='
+ expiryDate.toUTCString() + ';';
        response.setHeader('Set-Cookie', cookieText);
        response.writeHead(302, {
          'Location': '/'
        });

        return response.end();
    }
```

If we have set this cookie for the first time, the client is instructed to make another request to this same server. As there is now a cookie set for this domain, the subsequent request will contain our cookie, which we handle next:

```
     cookies.split(';').forEach(function(cookie) {
        var m = cookie.match(/(.*?)=(.*)$/);
        cookies[m[1].trim()] = (m[2] || '').trim();
    });
    response.end("Cookie set: " + cookies.toString());

}).listen(8080);
```

Understanding content types

A client will often pass along a request header indicating the expected response **MIME** (**Multi-purpose Internet Mail Extension**) type. Clients will also indicate the MIME type of a request body. Servers will similarly provide header information about the MIME type of a response body. The MIME type for HTML is **text/html**, for example.

As we have seen, it is the responsibility of an HTTP response to set headers describing the entity it contains. Similarly, a GET request will normally indicate the resource type, the MIME type, it expects as a response. Such a request header might look like this:

```
Accept: text/html
```

It is the responsibility of a server receiving such instructions to prepare a body entity conforming to the sent MIME type, and if it is able to do so it should return a similar response header:

```
Content-Type: text/html; charset=utf-8
```

Because requests also identify the specific resource desired (such as `/files/index.html`), the server must ensure that the requested resource it is streaming back to the client is in fact of the correct MIME type. While it may seem obvious that a resource identified by the extension `html` is in fact of the MIME type `text/html`, this is not at all certain—a filesystem does nothing to prevent an image file from being given an "html" extension. Parsing extensions is an imperfect method of determining file type. We need to do more.

The UNIX `file` program is able to determine the MIME type of a system file. For example, one might determine the MIME type of a file without an extension (for example, `resource`) by running this command:

```
file --brief --mime resource
```

We pass arguments instructing `file` to output the MIME type of resource, and that the output should be brief (only the MIME type, and no other information).

This command might return something like `text/plain; charset=us-ascii`. Here we have a tool to solve our problem.

For more information about the `file` utility consult go to the following link:

`http://unixhelp.ed.ac.uk/CGI/man-cgi?file`

Recalling that Node is able to spawn child processes, we have a solution to our problem of accurately determining the MIME type of system files.

We can use the Node command `exec` method of Node's `child_process` module in order to determine the MIME type of a file like so:

```
var exec = require('child_process').exec;
exec("file --brief --mime resource", function(err, mime) {
    console.log(mime);
});
```

This technique is also useful when validating a file streamed in from an external location. Following the axiom "never trust the client", it is always a good idea to check whether the `Content-type` header of a file posted to a Node server matches the actual MIME type of the received file as it exists on the local filesystem.

Handling favicon requests

When visiting a URL via a browser one will often notice a little icon in the browser tab or in the browser's address bar. This icon is an image, named `favicon.ico` and it is fetched on each request. As such, an HTTP GET request is normally combines two requests—one for the favicon, and another for the requested resource.

Node developers are often surprised by this doubled request. Any implementation of an HTTP server must deal with favicon requests. To do so, the server must check the request type, and handle it accordingly. The following example demonstrates one method of doing so:

```
var http = require('http');
http.createServer(function(request, response) {
    if(request.url === '/favicon.ico') {
        response.writeHead(200, {
            'Content-Type': 'image/x-icon'
        });
        return response.end();
    }
  response.writeHead(200, {
      'Content-Type': 'text/plain'
  });
  response.write('Some requested resource');
  response.end();

}).listen(8080);
```

This code will simply send an empty image stream for the favicon. If there is a favicon to send, one would simply push that data through the response stream, as we've discussed previously.

Handling POST data

One of the most common REST methods used in network applications is POST. According to the REST specification a POST is *not* idempotent, as opposed to most of the other well-known methods (GET, PUT, DELETE, and so on) that are. This is mentioned in order to point out that the handling of POST data will very often have a consequential effect on an application's state, and should therefore be handled with care.

We will now discuss handling of the most common type of POST data, that which is submitted via forms. The more complex type of POST—multipart uploads—will be discussed in *Chapter 4*, *Using Node to Access the Filesystem*.

Let's create a server which will return a form to clients, and echo back any data that client submits with that form. We will need to first check the request URL, determining if this is a form request or a form submission, returning HTML for a form in the first case, and parsing submitted data in the second:

```
var http = require('http');
var qs = require('querystring');
http.createServer(function(request, response) {
    var body = "";
    if(request.url === "/") {
        response.writeHead(200, {
            "Content-Type": "text/html"
        });
        return response.end(
            '<form action="/submit" method="post">\
            <input type="text" name="sometext">\
            <input type="submit" value="Upload">\
            </form>'
        );
    }
```

Note that the form we respond with has a single field named `sometext`. This form should POST data in the form `sometext=entered_text`:

```
    if(request.url === "/submit") {
        request.on('readable', function() {
            body += request.read();

        });
        request.on('end', function() {
            var fields = qs.parse(body);
            response.end("Thanks!");
```

```
            console.log(fields)
        });
    }
}).listen(8080);
```

Once our POST stream ends and the client is notified that the POST is received, we parse the posted data using `Querystring.parse`, giving us a Key/Value map accessible via `fields["somedata"]`.

Creating and streaming images with Node

Having gone over the main strategies for initiating and diverting streams of data, let's practice the theory by creating a service to stream (aptly named) **PNG** (**Portable Network Graphics**) images to a client. This will not be a simple file server, however. The goal is to create PNG data streams by piping the output stream of an **ImageMagick** `convert` operation executing in a separate process into the response stream of an HTTP connection, where the converter is translating another stream of **SVG** (**Scalable Vector Graphics**) data generated within a virtualized **DOM** (**Document Object Model**) existing in the Node runtime. Let's get started.

The full code for this example can be found in your code bundle.

Our goal is to use Node to generate pie charts dynamically on a server based on client requests. A client will specify some data values, and a PNG representing that data in a pie will be generated. We are going to use the **D3.js** library, which provides a Javascript API for creating data visualizations, and the **jsdom** NPM package, which allows us to create a virtual DOM within a Node process.

Additionally, the PNG we create will be written to a file. If future requests pass the same query arguments to our service, we will then be able to rapidly pipe the existing rendering immediately, without the overhead of regenerating it.

A pie graph represents a range if percentages whose sum fills the total area of a circle, visualized as slices. Our service will draw such a graph based on the values a client sends. In our system the client is required to send values adding up to 1, such as .5, .3, .2. Our server, when it receives a request, will therefore need to fetch query parameters as well as create a unique key that maps to future requests with the same query parameters:

```
    var values         = url.parse(request.url, true).query['values'].
split(",");
      var cacheKey     = values.sort().join('');
```

Here we see the URL module in action, pulling out our data values. As well, we create a key on these values by first sorting the values, then joining them into a string we will use as the filename for our cached pie graph. We sort values for this reason: the same graph is achieved by sending `.5 .3 .2` and `.3 .5 .2`. By sorting and joining these both become the filename `.2 .3 .5`.

In a production application, more work would need to be done to ensure that the query is well formed, is mathematically correct, and so on. In our example we assume proper values are being sent.

Creating, caching, and sending a PNG representation

Assuming we do not have a cached copy, we will need to create one. Our system works by creating a virtual DOM, using D3 to generate an SVG pie chart within that DOM, passing the generated SVG to the ImageMagick `convert` program, which converts SVG data into a PNG representation. Additionally, we will need to store the created PNG on a filesystem.

We use **jsdom** to create a DOM, and use D3 to create the pie chart. Visit `https://github.com/tmpvar/jsdom` to learn about how jsdom works, and `http://d3js.org/` to learn about using D3 to generate SVG. Assume that we have created this SVG, and stored it in a variable `svg`, which will contain a string similar to this:

```
<svg width="200" height="200">
    <g transform="translate(100,100)">
        <defs>
            <radialgradient id="grad-0" gradientUnits="userSpaceOnUse"
cx="0" cy="0" r="100">
                <stop offset="0" stop-color="#7db9e8"></stop>
...
```

We now must convert that SVG into a PNG. To do this we spawn a child process running the ImageMagick `convert` program, and stream our SVG data to the `stdin` of that process:

```
var svgToPng = spawn("convert", ["svg:", "png:-"]);
svgToPng.stdin.write(svg);
svgToPng.stdin.end();
```

The `stdout` of the `svgToPng` process (a `ReadableStream`) will push a byte stream representing the PNG of our pie graph. This stream will be piped to two `WritableStreams`: the response object, such that our client receives the PNG data, and a new PNG file, which filename will be stored in the variable `cacheKey`. A response stream already exists; we must now create the file stream:

```
var filewriter = fs.createWriteStream(cacheKey);
filewriter.on("open", function(err) {
...
```

Once we have successfully opened a file stream we have all our streams in place: a `ReadableStream` represented by the `stdout` of the convert process; a `WritableStream` bound to a file on our filesystem; and the `WritableStream` represented by our server's response object. We must now achieve this flow: PNG conversion stream > file stream > response stream:

```
svgToPng.stdout.pipe(file).pipe(response);
```

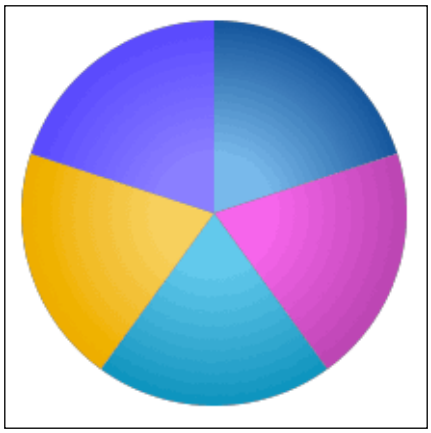
This will not work as is, however: a `WritableStream` (file) cannot be piped to another `WritableStream` (response), as such a stream does not push data. One way to solve this is to use a `TransformStream`:

```
var streamer = new stream.Transform();
streamer._transform = function(data, enc, cb) {
    filewriter.write(data);
    this.push(data);
    cb();
};
svgToPng.stdout.pipe(streamer).pipe(response);
```

Recalling our discussion on streams, what is happening here should be clear. A `TransformStream` functions as a `DuplexStream`, being both readable and writable. We override its abstract `_transform` method, and do our file writing there. Once we successfully write to a file, we then push the same data forward onto the response stream. We can now achieve the desired stream chaining:

```
svgToPng.stdout.pipe(streamer).pipe(response);
```

And the client receives a pie graph:



Finally, we need to handle cases where the requested pie chart has already been rendered and can be directly streamed from a filesystem:

```
fs.exists(cacheKey, function(exists) {
    response.writeHead(200, {
        'Content-Type': 'image/png'
    });

    if(exists) {
        fs.createReadStream(cacheKey)
        .on('readable', function() {
            var chunk;
            while(chunk = this.read()) {
                response.write(chunk);
            }
        })
        .on('end', function() {
            response.end();
        });

        return;
    }
    ...
```

Once we have determined that a cached file exists, we run a `while` loop pulling data off the file stream, writing this data to a response object, until the file stream ends.

Summary

As we have learned, Node's designers have succeeded in creating a simple, predictable, and convenient solution to the very difficult problem of enabling efficient I/O between disparate sources and targets. Its abstract `Stream` interface facilitates the instantiation of consistent readable and writable interfaces, and the extension of this interface into HTTP requests and responses, the filesystem, child processes, and other data channels makes stream programming with Node a pleasant experience.

Now that we've learned how to set up HTTP servers to handle streams of data arriving from many simultaneously connected clients, and how to feed those clients buffets of buffered streams, we can begin to engage more deeply with the task of building enterprise-grade concurrent real-time systems with Node.

4

Using Node to Access the Filesystem

We have persistent objects — they're called files.

— *Ken Thompson*

A file is simply a chunk of data that is persisted, usually, on some hard medium such as a hard drive. Files are normally composed of a sequence of bytes whose encoding maps onto some other pattern, like a sequence of numbers or electrical pulses. A nearly infinite number of encodings are possible, with some common ones being text files, image files, and music files. Files have a fixed length, and to be read their character encoding must be deciphered by some sort of reader, like an MP3 player or a word processor.

When a file is in transit, moving through a cable after it's been siphoned off of some storage device, it is no different than any other data stream running through the wire. Its previous solid state is just a stable blueprint that can be easily and infinitely copied, sliced up, and reworked.

We've seen how evented streams reflect the core design principles informing Node's design, where byte streams are to be read from and written to, and piped into other streams, emitting relevant stream events, such as end. Files are easily understood as being containers of data, filled with bytes that can be extracted or inserted partially or as a whole.

In addition to their natural similarity to streams, files also display the characteristics of objects. Files have properties that describe the interface available for accessing file contents—data structures with properties and associated access methods.

A **filesystem** reflects some concept of how files should be organized—how they are identified, where they are stored, how they are to be accessed, and so forth. A common filesystem for UNIX users is the **UFS** (**Unix File System**), while Windows users may be familiar with **NTFS** (**New Technology File System**).

It is interesting that the designers of the Plan 9 operating system (a team including *Ken Thompson*) decided to have all control interfaces represented as filesystems, such that all system interfaces (across devices, across applications) are modeled as file operations. Treating files as first-class citizens is a philosophy the UNIX OS also uses—using files as references to named pipes and sockets, among other things, gives developers enormous power when shaping data flow.

File objects are also powerful, and the system they reside within exposes fundamental I/O interfaces that must be easy to use, consistent, and very fast. Not surprisingly, Node's `file` module exposes just such an interface.

We will be considering handling files in Node from these two perspectives: how file data content is streamed in and out (read from and written), and how the attributes of file objects are modified, such as changing file permissions.

Additionally, we will cover the responsibilities of the Node server, in terms of accepting file uploads and servicing file requests. By working through examples of directory iterators and file servers the full range and behavior of Node's filesystem API should become clear.

For those with knowledge of C++ and a healthy curiosity, the filesystem implementation in Node can be browsed here: `https://github.com/joyent/node/blob/master/src/node_file.cc`.

Directories, and iterating over files and folders

Improving Node's performance with regards to the filesystem has been one of the main challenges taken up by the core Node team. For example, a (flawed) negative comparison to the Vert.x environment's file serving speed led to some interesting commentary from developers and users of both systems which, while not all constructive, is a good read if one is interested in how filesystems are accessed by Node and other systems: `http://vertxproject.wordpress.com/2012/05/09/vert-x-vs-node-js-simple-http-benchmarks/`.

Typically, a filesystem groups files into collections, normally referred to as directories. One navigates through directories to find individual files. Once a target file is found, the file object must be wrapped by an interface exposing the file contents for reading and writing.

Because Node development often involves the creation of servers that both accept and emit file data, it should be clear how important transfer speed at this active and important I/O layer is. As mentioned earlier, files can also be understood as objects, and objects have certain attributes.

Types of files

There are six types of files commonly encountered on a UNIX system:

- **Ordinary files**: These contain a one-dimensional array of bytes, and cannot contain other files.
- **Directories**: These are also files, implemented in a special way such that they can describe collections of other files.
- **Sockets**: Used for IPC, allowing processes to exchange data.
- **Named pipe**: A command such as `ps aux | grep node` creates a pipe, which is destroyed once the operation terminates. Named pipes are persistent and addressable, and can be used variously by multiple processes for IPC indefinitely.
- **Device files**: These are representations of I/O devices, processes that accept streams of data. `/dev/null` is commonly an example of a character device file (accepts serial streams of I/O), and `/dev/sda` is an example of a block device file (allowing random access I/O for blocks of data) representing a data drive.
- **Links**: These are pointers to other files of two types, hard links and symbolic links. Hard links directly point to another file, and are indistinguishable from the target file. Symbolic links are indirect pointers, and are distinguishable from normal files.

Most Node filesystem interactions encounter only the first two types, with the third only indirectly via the Node API. A deeper explanation of the remaining types is beyond the scope of this discussion. However, Node provides a full suite of file operations via the `file` module, and the reader should have at least some familiarity with the full range and power of file types.

Studying named pipes will reward the reader interested in understanding how Node was designed to work with streams and pipes. Try this from a terminal:

```
mkfifo namedpipe
```

Following `ls -l` a listing similar to this will be shown:

```
prw-r--r--  1 system  staff    0 May 01 07:52 namedpipe
```

Note the `p` flag in the file mode. You've created a named pipe. Now enter this into the same terminal, pushing some bytes into the named pipe:

```
echo "hello" > namedpipe
```

It will seem like the process has hung. It hasn't—pipes, like water pipes, must be open on both ends to complete their job of flushing contents. We've pumped some bytes in… now what?

Open another terminal, navigate to the same directory, and enter:

```
cat namedpipe
```

`hello` will appear as the contents of `namedpipe` are flushed. Note as well that the first terminal is no longer hung.

File paths

Most of the filesystem methods provided by Node will require the manipulation of file paths, and for this purpose we make use of the `path` module. We can compose, decompose, and relate paths with this module. Instead of hand rolling your own path string splitting and regexing and concatenating routines, try to normalize your code by delegating path manipulation to this module.

- Use `path.normalize` whenever working with a file path string whose source is untrusted or unreliable, to ensure a predictable format:

```
var path = require('path');
path.normalize("../one////two/./three.html");
// -> ../one/two/three.html
```

- Use `path.join` whenever building a single path out of path segments:

```
path.join("../","one","two","three.html");
// -> ../one/two/three.html
```

- Use `path.dirname` to snip the directory name out of a path:

```
path.dirname("../one/two/three.html");
// ../one/two
```

- Use `path.basename` to manipulate the final path segment:

```
path.basename("../one/two/three.html");
// -> three.html

// Remove file extension from the basename
path.basename("../one/two/three.html", ".html");
// -> three
```

- Use `path.extname` to slice from the last period(.) to the end of the path string:

```
var pstring = "../one/two/three.html";
path.extname(pstring);
// -> .html
//
// Which is identical to:
// pstring.slice(pstring.lastIndexOf("."));
```

- Use `path.relative` to find the relative path from one absolute path to another:

```
path.relative(
  '/one/two/three/four',
  '/one/two/thumb/war'
);
// -> ../../thumb/war
```

- Use `path.resolve` to resolve a list of path instructions into an absolute path:

```
path.resolve('/one/two', '/three/four');
// -> /three/four
path.resolve('/one/two/three', '../', 'four', '../../five')
// -> /one/five
```

Think of the arguments passed to `path.resolve` as being a sequence of `cd` calls:

```
cd /one/two/three
cd ../
cd four
cd ../../five
pwd
// -> /one/five
```

If the list of arguments passed to `path.resolve` fails to deliver an absolute path, the current directory name is used as well. For instance, if we are in `/users/home/john/`:

```
path.resolve('one','two/three','four');
// -> /users/home/john/one/two/three/four
```

These arguments resolve to a relative path `one/two/three/four` that is therefore prefixed with the current directory name.

File attributes

A file object exposes some of its attributes, comprising a useful set of metadata about the file data. If one is using Node to run an HTTP server it will be necessary to determine the file length of any file requested via a GET, for example. Determining the time a file was last modified finds uses across many types of applications.

To read the attributes of a file, use `fs.stat`:

```
fs.stat("file.txt", function(err, stats) {
  console.log(stats);
});
```

In the preceding example, `stats` will be an `fs.Stats` object describing the file through a map of attributes:

```
  dev: 2051, // id of device containing this file
  mode: 33188, // bitmask, status of the file
  nlink: 1, // number of hard links
  uid: 0, // user id of file owner
  gid: 0, // group id of file owner
  rdev: 0, // device id (if device file)
  blksize: 4096, // I/O block size
  ino: 27396003, // a unique file inode number
  size: 2000736, // size in bytes
  blocks: 3920, // number of blocks allocated
  atime: Fri May 3 2013 15:39:57 GMT-0500 (CDT), // last access
  mtime: Fri May 3 2013 17:22:46 GMT-0500 (CDT), // last modified
  ctime: Fri May 3 2013 17:22:46 GMT-0500 (CDT)  // last status change
```

An `fs.Stats` object exposes several useful methods for accessing file attribute data:

- Use `stats.isFile` to check for standard files
- Use `stats.isDirectory` to check for directories
- Use `stats.isBlockDevice` to check for block type device files
- Use `stats.isCharacterDevice` to check for character type device files

- Use `stats.isSymbolicLink` after an `fs.lstat` to find symbolic links
- Use `stats.isFIFO` to identify named pipes
- Use `stats.isSocket` to check for sockets

There are two further `stat` methods available:

- `fs.fstat(fd, callback)`: Like `fs.stat`, except that a file descriptor `fd` is passed rather than a file path
- `fs.lstat(path, callback)`: An `fs.stat` on a symbolic link will return an `fs.Stats` object for the target file, while `fs.lstat` will return an `fs.Stats` object for the link file itself

The following two methods simplify the file timestamp manipulation:

- `fs.utimes(path, atime, mtime, callback)`: Change the access and modify timestamps on a file at `path`. The access and modify times of a file are stored as instances of the JavaScript `Date` object. `Date.getTime` will, for example, return the number of milliseconds elapsed since midnight (UTC) on January 1, 1970.
- `fs.futimes(fd, atime, mtime, callback)`: Changes the access and modify timestamps on a file descriptor `fd`. Similar to `fs.utimes`.

More information about manipulating dates and times with JavaScript can be found here: `https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Date`.

Opening and closing files

One of the unofficial rules governing the Node project is to not unnecessarily abstract away from existing OS implementation details. As we will see, references to file descriptors appear throughout Node's file API. For **POSIX** (**Portable Operating System Interface**), a file descriptor is simply an (non-negative) integer uniquely referencing a specific file. Since Node modeled its filesystem methods on POSIX, not surprisingly a file descriptor is represented in Node as an integer.

Recalling our discussion of how devices and other elements of the OS are represented as files, it would stand to reason that the standard I/O streams (`stdin`, `stdout`, `stderr`) would also have file descriptors. In fact, that is the case:

```
console.log(process.stdin.fd); // 0
console.log(process.stdout.fd); // 1
console.log(process.stderr.fd); // 2
```

```
fs.fstat(1, function(err, stat) {
  console.log(stat); // an fs.Stats object
});
```

File descriptors are easy to obtain, and convenient ways to pass around file references. Let's look at how file descriptors are created and used by examining how to perform low level file open and close operations using Node. As the chapter progresses we'll investigate more refined interfaces to file streams.

fs.open(path, flags, [mode], callback)

Trying to open a file at `path`. `callback` will receive any exceptions with the operation as its first argument, and a file descriptor as its second argument. Here we open a file for reading:

```
fs.open("path.js", "r", function(err, fileDescriptor) {
  console.log(fileDescriptor); // An integer, like `7` or `23`
});
```

`flags` receives a string indicating the types of operations the caller expects to perform on the returned file descriptor. Their meanings should be clear:

- **r**: Opening file for reading, throwing an exception if the file doesn't exist.
- **r+**: Opening file for both reading and writing, throwing an exception if the file doesn't exist.
- **w**: Opening file for writing, creating the file if it doesn't exist, and truncating the file to zero bytes if it does exist.
- **wx**: Like **w**, but opens the file in exclusive mode, which means if the file already exists it will **not be opened**, and the open operation will fail. This is useful if multiple processes may be simultaneously trying to create the same file.
- **w+**: Opening file for reading and writing, creating the file if it doesn't exist, and truncating the file to zero bytes if it does exist.
- **wx+**: Like **wx** (and **w**), additionally opening the file for reading.
- **a**: Opening file for appending, creating the file if it does not exist.
- **ax**: Like **a**, but opens the file in exclusive mode, which means if the file already exists it will **not be opened**, and the open operation will fail. This is useful if multiple processes may be simultaneously trying to create the same file.
- **a+**: Open file for reading and appending, creating the file if it does not exist.
- **ax+**: Like **ax** (and **a**), additionally opening the file for reading.

When an operation may create a new file, use the optional `mode` to set permissions for this file in octal digits, defaulting to 0666 (see `fs.chmod` for more information about octal permissions):

```
fs.open("index.html","w", 755, function(err, fd) {
   fs.read(fd, …);
});
```

fs.close(fd, callback)

The `fs.close(fd, callback)` method closes a file descriptor. The callback receives one argument, any exception thrown in the call. It's a good habit to close all file descriptors that have been opened.

File operations

Node implements the standard POSIX functions for working with files, which a UNIX user will be familiar with. We will not be covering each member of this extensive collection in depth, instead focusing on some commonly used examples. In particular we will be going into depth discussing the methods for opening file descriptors and manipulating file data, reading and manipulating file attributes, and moving through filesystem directories. Nevertheless, the reader is encouraged to experiment with the entire set, which the following list briefly describes. Note that all of these methods are asynchronous, non-blocking file operations.

fs.rename(oldName, newName, callback)

The `fs.rename(oldName, newName, callback)` method renames file at `oldName` to `newName`. The callback receives one argument, any exception thrown in the call.

fs.truncate(path, len, callback)

The `fs.truncate(path, len, callback)` method changes the length of the file at `path` by `len` bytes. If `len` represents a length shorter than the file's current length, the file is truncated to that length. If `len` is greater, the file length is padded by appending null bytes (\x00) until `len` is reached. The callback receives one argument, any exception thrown in the call.

fs.ftruncate(fd, len, callback)

The `fs.ftruncate(fd, len, callback)` method is like `fs.truncate`, except that instead of specifying a file, a file descriptor is passed as `fd`.

fs.chown(path, uid, gid, callback)

The `fs.chown(path, uid, gid, callback)` method changes the ownership of the file at `path`. Use this to set whether user `uid` or group `gid` has access to a file. The callback receives one argument, any exception thrown in the call.

fs.fchown(fd, uid, gid, callback)

The `fs.fchown(fd, uid, gid, callback)` method is like `fs.chown`, except that instead of specifying a file path, a file descriptor is passed as `fd`.

fs.lchown(path, uid, gid, callback)

The `fs.lchown(path, uid, gid, callback)` method is like `fs.chown`, except that in the case of symbolic links ownership of the link file itself is changed, but not the referenced link.

fs.chmod(path, mode, callback)

The `fs.chmod(path, mode, callback)` method changes the `mode` (permissions) on a file at `path`. You are setting the read(4), write(2), and execute(1) bits for this file, which can be sent in octal digits:

	[r]ead	[w]rite	E[x]ecute	Total
Owner	4	2	1	7
Group	4	0	1	5
Other	4	0	1	5
				chmod(755)

You may also use symbolic representations, such as *g+rw* for group read and write, similar to the arguments we saw for `file.open` earlier. For more information on setting file modes, consult: `http://en.wikipedia.org/wiki/Chmod`.

The callback receives one argument, any exception thrown in the call.

fs.fchmod(fd, mode, callback)

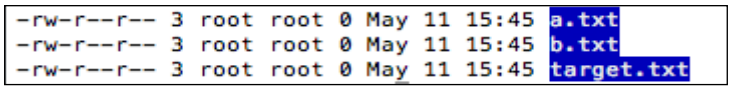
The `fs.fchmod(fd, mode, callback)` method is like `fs.chmod`, except that instead of specifying a file path, a file descriptor is passed as `fd`.

fs.lchmod(path, mode, callback)

The `fs.lchmod(path, mode, callback)` method is like `fs.chmod`, except that in the case of symbolic links permissions on the link file itself is changed, but not those of the referenced link.

fs.link(srcPath, dstPath, callback)

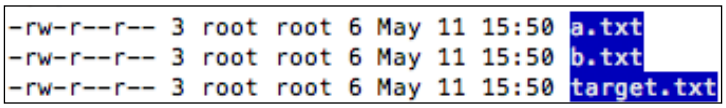
The `fs.link(srcPath, dstPath, callback)` creates a hard link between `srcPath` and `dstPath`. This is a way of creating many different paths to exactly the same file. For example, the following directory contains a file `target.txt` and two hard links—`a.txt` and `b.txt`—which each point to this file:



Note that `target.txt` is empty. If the content of the target file is changed, the length of the link files will also be changed. Changing the content of the target file:

```
echo "hello" >> target.txt
```

This results in this new directory structure, clearly demonstrating the hard references:



The callback receives one argument, any exception thrown in the call.

fs.symlink(srcPath, dstPath, [type], callback)

The `fs.symlink(srcPath, dstPath, [type], callback)` method creates a symbolic link between `srcPath` and `dstPath`. Unlike hard links created with `fs.link`, symbolic links are simply pointers to other files, and do not themselves respond to changes in the target file. The default link `type` is file. Other options are directory and junction, the last being a Windows-specific type that is ignored on other systems. The callback receives one argument, any exception thrown in the call.

Compare and contrast the directory changes described in our `fs.link` discussion to the following:

```
lrwxrwxrwx  1 root root   10 May 11 16:11 a.txt -> target.txt
lrwxrwxrwx  1 root root   10 May 11 16:11 b.txt -> target.txt
-rw-r--r--  1 root root    0 May 11 16:12 target.txt
```

Unlike hard links, symbolic links do not change in length when their target file (in this case `target.txt`) changes length. Here we see how changing the target's length from zero bytes to six bytes has no effect on the length of any bound symbolic links:

```
lrwxrwxrwx  1 root root   10 May 11 16:11 a.txt -> target.txt
lrwxrwxrwx  1 root root   10 May 11 16:11 b.txt -> target.txt
-rw-r--r--  1 root root    6 May 11 16:12 target.txt
```

fs.readlink(path, callback)

The given symbolic link at `path`, returns the filename of the targeted file:

```
fs.readlink('a.txt', function(err, targetFName) {
  console.log(targetFName); // target.txt
});
```

fs.realpath(path, [cache], callback)

The `fs.realpath(path, [cache], callback)` method attempts to find the real path to file at `path`. This is a useful way to find the absolute path to a file, resolve symbolic links, and even clean up extraneous slashes and other malformed paths. For example:

```
fs.realpath('file.txt', function(err, resolvedPath) {
  console.log(resolvedPath); // `/real/path/to/file.txt`
});
```

Or, even:

```
fs.realpath('./////./file.txt', function(err, resolvedPath) {
  // still `/real/path/to/file.txt`
});
```

If some of the path segments to be resolved are already known, one can pass a `cache` of mapped paths:

```
var cache = {'/etc':'/private/etc'};
fs.realpath('/etc/passwd', cache, function(err, resolvedPath) {
  console.log(resolvedPath); // `/private/etc/passwd`
});
```

fs.unlink(path, callback)

The `fs.unlink(path, callback)` method removes the file at `path`—equivalent to deleting a file. The callback receives one argument, any exception thrown in the call.

fs.rmdir(path, callback)

The `fs.rmdir(path, callback)` method removes the directory at `path`. Equivalent to deleting a directory.

Note that if the directory is not empty this will throw an exception. The callback receives one argument, any exception thrown in the call.

fs.mkdir(path, [mode], callback)

The `fs.mkdir(path, [mode], callback)` method creates a directory at `path`. To set the mode of the new directory, use the permission bit map described in `fs.chmod`.

Note that if this directory already exists, an exception will be thrown. The callback receives one argument, any exception thrown in the call.

fs.exists(path, callback)

The `fs.exists(path, callback)` method checks whether a file exists at `path`. The callback will receive a Boolean true or false.

fs.fsync(fd, callback)

Between the instant a request for some data to be written to a file is made and that data fully exists on a storage device, the candidate data exists within core system buffers. This latency isn't normally relevant but, in some extreme cases, such as system crashes, it is necessary to insist that the file reflect a known state on a stable storage device.

`fs.fsync` copies all in-core data of a file referenced by file descriptor `fd` to disk (or other storage device). The callback receives one argument, any exception thrown in the call.

Synchronicity

Conveniently, Node's `file` module provides synchronous counterparts for each of the asynchronous methods we've covered, indicated by the suffix `Sync`. For example, the synchronous version of `fs.mkdir` is `fs.mkdirSync`.

A synchronous call is also able to directly return its result, obviating the need for callbacks. While demonstrating the creation of HTTPS servers in *Chapter 3*, *Streaming Data Across Nodes and Clients*, we saw both a good use case for synchronous code and an example of direct assignment of results without a callback:

```
key: fs.readFileSync('server-key.pem'),
cert: fs.readFileSync('server-cert.pem')
```

Hey! Doesn't Node strictly enforce asynchronous programming? Isn't blocking code always wrong? All developers are encouraged to adhere to non-blocking designs, and you are encouraged to avoid synchronous coding—if facing a problem where a synchronous operation seems the only solution, it is likely that the problem has been misunderstood. Nevertheless, edge cases requiring a file object existing fully in memory prior to executing further instructions (a blocking operation) do exist. Node gives a developer the power to break with asynchronous tradition if it is the only possible solution (which it probably isn't!).

One synchronous operation developers regularly use (perhaps without realizing it) is the `require` directive:

```
require('fs')
```

Until the dependency targeted by require is fully initialized, subsequent JavaScript instructions will not execute (file loading blocks the event loop). *Ryan Dahl* struggled with this decision to introduce synchronous operations (in particular file operations) into Node, as he mentioned at a Google Tech Talk on July 9, 2013 (`http://www.youtube.com/watch?v=F6k8lTrAE2g`):

> *I think this is an OK compromise. It pained me for months, to drop the purity of having an asynchronous module system. But, I think it's ok.*

And later:

> *It simplifies the code a lot to be able to just stick in "require, require, require" and not have to do an onload callback…I think that's been a relatively OK compromise. […] There's really two parts to your program: there's the loading and starting up phase…and you don't really care how fast that runs...you're going to load modules and stuff…the setup phase of your daemon, generally, is synchronous. It's when you get into your event loop for serving requests that you want to be very careful about this. […] I will give people synchronous file I/O. If they do it in servers…it won't be terrible, right? The important thing is to never let them do synchronous network I/O.*

Synchronous code does have the advantage of being eminently predictable, as nothing else happens until this instruction is completed. When starting up a server, which will happen only rarely, Dahl is suggesting that a little certainty and simplicity goes a long way. The loading of configuration files, for example, might make sense on server initialization.

Sometimes a desire to use synchronous commands in Node development is simply a cry for help; a developer being overwhelmed by deeply nested callback structures. If ever faced with this pain, try some of the callback-taming libraries mentioned in *Chapter 2, Understanding Asynchronous Event-Driven Programming*.

Moving through directories

Let's apply what we have learned and create a directory iterator. The goal for this project is to create a function that will accept a directory path and return a JSON object reflecting the directory hierarchy of files, its nodes composed of file objects. We will also make our directory walker a more powerful event-based parser, consistent with the Node philosophy.

To move through nested directories one must first be able to read a single directory. Node's filesystem library provides the `fs.readdir` command for this purpose:

```
fs.readdir('.', function(err, files) {
  console.log(files); // list of all files in current directory
});
```

Remembering that everything is a file, we will need to do more than simply get a directory listing; we must determine the type of each member of our file list. By adding `fs.stat` we have already completed a large majority of the logic:

```
(function(dir) {
  fs.readdir(dir, function(err, list) {
    list.forEach(function(file) {
      fs.stat(dir + "/" + file, function(err, stat) {
        if(stat.isDirectory()) {
          return console.log("Found directory : " + file);
        }
        console.log("Found file : " + file);
      });
    });
  });
})(".");
```

This self-executing function receives a directory path argument (`"."`), folds that directory listing into an array of file names, fetches an `fs.Stats` object for each of these, and makes a decision based on the indicated file type (directory or not a directory) on what to do next. At this point we also have available to us the name of the current file and its attributes. Clearly, we have already mapped a single directory.

We must now map directories within directories, storing results in a JSON object reflecting the nested filesystem tree, each leaf on the tree a file object. Recursively passing our directory reader function paths to subdirectories and appending returned results as branches of the final object is the next step:

```
var walk = function(dir, done) {
  var results = {};
  fs.readdir(dir, function(err, list) {
    var pending = list.length;
    if(err || !pending) {
      return done(err, results);
    }
    list.forEach(function(file) {
      var dfile = dir + "/" + file;
      fs.stat(dfile, function(err, stat) {
        if(stat.isDirectory()) {
          return walk(dfile, function(err, res) {
            results[file] = res;
            !--pending && done(null, results);
          });
        }
        results[file] = stat;
        !--pending && done(null, results);
      });
    });
  });
};
walk(".", function(err, res) {
  console.log(require('util').inspect(res, {depth: null}));
});
```

We create a `walk` method which receives a directory path and a callback that receives the directory graph or an error when `walk` is complete, following Node's style. Not much coding is needed to create a very fast, non-blocking file tree walker, complete with file stats.

Now let's publish events whenever a directory or file is encountered, giving any future implementation flexibility to construct its own representation of the filesystem. To do this we will use the friendly `EventEmitter` object:

```
var walk = function(dir, done, emitter) {
  ...
  emitter = emitter || new (require('events').EventEmitter);
  ...
  if(stat.isDirectory()) {
    emitter.emit('directory', dfile, stat);
    return walk(dfile, function(err, res) {
      results[file] = res;
      !--pending && done(null, results);
    }, emitter);
  }
  emitter.emit('file', dfile, stat);
  results[file] = stat;
  ...
  return emitter;
}
walk("/usr/local", function(err, res) {
  ...
}).on("directory", function(path, stat) {
  console.log("Directory: " + path + " - " + stat.size);
}).on("file", function(path, stat) {
  console.log("File: " + path + " - " + stat.size);
});
// File: index.html – 1024
// File: readme.txt – 2048
// Directory: images - 106
// File images/logo.png - 4096
// ...
```

Now that we know how to discover and address files, we can start reading from and writing to them.

Reading from a file

In our discussion of file descriptors we touched on one method of opening a file, fetching a file descriptor, and ultimately pushing or pulling data through that reference. Reading files is a common operation. Sometimes managing a read buffer precisely may be necessary, and Node allows byte-by-byte control. In other cases one simply wants a no-frills stream that is simple to use.

Reading byte by byte

The `fs.read` method is the most low-level way Node offers for reading files.

fs.read(fd, buffer, offset, length, position, callback)

Files are composed of ordered bytes, and these bytes are addressable by their `position`, relative to the beginning of in the file (position zero [0]). Once we have a file descriptor `fd`, we can begin to read `length` number of bytes and insert those into a `Buffer` object `buffer`, insertion beginning at a given buffer `offset`. For example, to copy the 8366 bytes beginning at `position` 309 of readable file `fd` into a `buffer` beginning at an `offset` of 100, we would use: `fs.read(fd, buffer, 100, 8366, 309, callback)`.

The following code demonstrates how to open and read a file in 512 byte chunks:

```
fs.open('path.js', 'r', function(err, fd) {
    fs.fstat(fd, function(err, stats) {
        var totalBytes = stats.size;
        var buffer    = new Buffer(totalBytes);
        var bytesRead  = 0;
    //  Each call to read should ensure that chunk size is
    //  within proper size ranges (not too small; not too large).
    var read = function(chunkSize) {
        fs.read(fd, buffer, bytesRead, chunkSize, bytesRead,
function(err, numBytes, bufRef) {

          if((bytesRead += numBytes) < totalBytes) {
            return read(Math.min(512, totalBytes - bytesRead));
          }
          fs.close(fd);
          console.log("File read complete. Total bytes read: " +
totalBytes);
             // Note that the callback receives a reference to the
             // accumulating buffer
          console.log(bufRef.toString());
        });
    }

    read(Math.min(512, totalBytes));
    });
});
```

The resulting buffer can be piped elsewhere (including a server response object). It can also be manipulated using the methods of Node's `Buffer` object, such as conversion into a UTF8 string with `buffer.toString("utf8")`.

Fetching an entire file at once

Often one simply needs to fetch an entire file, without any ceremony or fine control. Node provides a shortcut method for exactly this.

fs.readFile(path, [options], callback)

Fetching the data contained by file `path` can be accomplished in one step:

```
fs.readFile('/etc/passwd', function(err, fileData) {
  if(err) {
    throw err;
  }
  console.log(fileData);
  // <Buffer 48 65 6C 6C 6F … >
});
```

We see how `callback` receives a buffer. It may be more desirable to receive the file data in a common encoding, such as UTF8. We are able to specify the encoding of the returned data, as well as the read mode, using the `options` object, which has two possible attributes:

- **encoding**: A string, such as `utf8`. Defaults to null (no encoding).
- **flag**: The file mode as a string. Defaults to `r`.

Modifying the previous example:

```
fs.readFile('/etc/passwd', function(err, { encoding : "utf8" },
fileData) {
  ...
  console.log(fileData);
  // "Hello ..."
});
```

Creating a readable stream

While `fs.readFile` is an excellent, simple way to accomplish a common task, it does have the significant drawback of requiring that an entire file be read into memory prior to any part of the file being sent to a callback. For large files, especially regularly accessed large files (such as videos), this is problematic as we are unable to accurately predict the volume of data we will be buffering at any given point in time. This means the amount of memory available to Node may be starved unpredictably, taking down an entire application or, worse, make your customers angry as responsiveness declines and errors begin to surface.

In the previous chapter we learned about data streams and the `Stream` object. While files are easily and naturally handled using readable streams, Node provides a dedicated file streaming interface which offers a compact file streaming facility without the extra construction work, with more flexibility than that offered by `fs.readFile`.

fs.createReadStream(path, [options])

The `fs.createReadStream(path, [options])` method returns a readable stream object for file at `path`. You may then perform stream operations on the returned object, such as `pipe()`.

The following options are available:

- **flags**: File mode argument as a string. Defaults to `r`.
- **encoding**: One of `utf8`, `ascii`, or `base64`. Default to no encoding.
- **fd**: One may set `path` to null, instead passing the call a file descriptor.
- **mode**: Octal representation of file mode, defaulting to 0666.
- **bufferSize**: The chunk size, in bytes, of the internal read stream. Defaults to 64 * 1024 bytes. You can set this to any number, but memory allocation is strictly controlled by the host OS, which may ignore a request. See: `https://groups.google.com/forum/?fromgroups#!topic/nodejs/p5FuU1oxbeY`.
- **autoClose**: Whether to automatically close the file descriptor (a la `fs.close`). Defaults to true. You may want to set this to false and close manually if you are sharing a file descriptor across many streams, as closing a descriptor will disrupt any other readers.
- **start**: Begin reading from this position. Default is 0.
- **end**: Stop reading at this position. Default is the file byte length.

Reading a file line by line

While reading a file stream byte-by-byte is sufficient for any file-parsing job, text files in particular are often more usefully read line by line, such as when reading logfiles. More precisely, any stream can be understood in terms of the chunks of data separated by newline characters, typically "\r\n" on UNIX systems. Node provides a native module whose methods simplify access to newline-separated chunks in data streams.

The Readline module

The `Readline` module has a simple but powerful goal, that is, to make reading a stream of data line-by-line easier. The bulk of its interface is designed to make command-line prompting easier, such that interfaces taking user input are easier to design.

Remembering that Node is designed for I/O, that I/O operations normally involve moving data between readable and writable streams, and that `stdout` and `stdin` are stream interfaces identical with the file streams returned by `fs.createReadStream` and `fs.createWriteStream`, we will look at how this module can be similarly used to prompt file streams for a line of text.

The `Readline` module will be used throughout the book (and later in this chapter), for purposes beyond the present discussion on reading files, such as when designing command-line interfaces. For the impatient, more information on this module can be found at: `http://nodejs.org/api/readline.html`.

To start working with the `Readline` module one must create an interface defining the input stream and the output stream. The default interface options prioritize usage as a terminal interface. The options we are interested in are:

- **input**: Required. The readable stream being listened to.
- **output**: Required. The writable stream being written to.
- **terminal**: Set this to true if both the input and output streams should be treated like a Unix terminal, or **TTY** (**TeleTYpewriter)**. For files, you will set this to false.

Reading the lines of a file is made easy through this system. For example, assuming one has a dictionary file listing common words in the English language one might want to read the list into an array for processing:

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

var rl = readline.createInterface({
  input: fs.createReadStream("dictionary.txt"),
  terminal: false
});
var arr = [];
rl.on("line", function(ln) {
   arr.push(ln.trim())
});
// aardvark
```

```
// abacus
// abaisance
// ...
```

Note how we disable TTY behavior, handling the lines ourselves without redirecting to an output stream.

As expected with a Node I/O module, we are working with stream events. The events listeners that may be of interest are:

- **line**: Receives the most recently read line, as a string
- **pause**: Called whenever the stream is paused
- **resume**: Called whenever a stream is resumed
- **close**: Called whenever a stream is closed

Except for **line**, these event names reflect `Readline` methods, pause a stream with `Readline.pause`, **resume** with `Readline.resume`, and **close** with `Readline.close`.

Writing to a file

As with reading files, Node provides a rich collection of tools for writing to files. We'll see how Node makes it as easy to target a file's contents byte-by-byte, as it is to pipe continuous streams of data into a single writable file.

Writing byte by byte

The `fs.write` method is the most low-level way Node offers for writing files. This method gives us precise control over where bytes will be written to in a file.

fs.write(fd, buffer, offset, length, position, callback)

To write the collection of bytes between positions 309 and 8675 (`length` 8366) of `buffer` to the file referenced by file descriptor `fd`, insertion beginning at `position` 100:

```
var buffer = new Buffer(8675);
fs.open("index.html", "w", function(err, fd) {
  fs.write(fd, buffer, 309, 8366, 100, function(err, writtenBytes,
buffer) {
    console.log("Wrote " + writtenBytes + " bytes to file");
    // Wrote 8366 bytes to file
  });
});
```

Note that for files opened in append (`a`) mode some operating systems may ignore `position` values, always adding data to the end of the file. Additionally, it is unsafe to call `fs.write` multiple times on the same file without waiting for the callback. Use `fs.createWriteStream` in those cases.

With such precise control we can intelligently structure files. In the following (somewhat contrived) example we create a file-based database containing indexed information for six months of baseball scores for a single team. We want to be able to quickly look up whether this team won or lost (or did not play) on a given day.

Since a month can have at most 31 days, we can (randomly) create a 6 x 31 grid of data in this file, placing one of three values in each grid cell: L (loss), W (win), N (no game). For fun, we also create a simple **CLI** (**Command-Line Interface**) to our database with a basic query language. This example should make it clear how `fs.read`, `fs.write` and `Buffer` objects are used to precisely manipulate bytes in files:

```
var fs     = require('fs');
var readline   = require('readline');
var cells  = 186; // 6 x 31
var buffer = new Buffer(cells);
while(cells--) {
  //  0, 1 or greater
  var rand = Math.floor(Math.random() * 3);
  //  78 = "N", 87 = "W", 76 = "L"
  buffer[cells] = rand === 0 ? 78 : rand === 1 ? 87 : 76;
}
fs.open("scores.txt", "r+", function(err, fd) {
  fs.write(fd, buffer, 0, buffer.length, 0, function(err,
writtenBytes, buffer) {
    var rl = readline.createInterface({
      input: process.stdin,
      output: process.stdout
    });

    var quest = function() {
      rl.question("month/day:", function(index) {
        if(!index) {
          return rl.close();
        }
        var md   = index.split('/');
        var pos = parseInt(md[0] -1) * 31 + parseInt(md[1] -1);
        fs.read(fd, new Buffer(1), 0, 1, pos, function(err, br, buff)
{
          var v = buff.toString();
```

```
            console.log(v === "W" ? "Win!" : v === "L" ? "Loss..." : "No
game");
            quest();
          });
        });
      };
      quest();
    });
});
```

Once running, we can simply type in a month/day pair and rapidly access that data cell. Adding in bounds checking for the input values would be a simple improvement. Pushing the file stream through a visualizing UI might be a nice exercise.

Writing large chunks of data

For straightforward write operations `fs.write` may be overkill. Sometimes all that is needed is a way to create a new file with some content. Just as common is the need to append data to the end of a file, as one might do in a logging system. The `fs.writeFile` and `fs.appendFile` methods can help us with those scenarios.

fs.writeFile(path, data, [options], callback)

The `fs.writeFile(path, data, [options], callback)` method writes the contents of `data` to the file at `path`. The `data` argument can be either a `Buffer` or a string. The following options are available:

- **encoding**: Defaults to `utf8`. If data is a buffer this option is ignored.
- **mode**: Octal representation of file mode, defaulting to 0666.
- **flag**: Write flags, defaulting to `w`.

Usage is straightforward:

```
fs.writeFile('test.txt', 'A string or Buffer of data', function(err) {
  if(err) {
    return console.log(err);
  }
  // File has been written
});
```

fs.appendFile(path, data, [options], callback)

Similar to `fs.writeFile`, except that `data` is appended to the end of the file at `path`. As well, the `flag` option defaults to `a`.

Creating a writable stream

If the data being written to a file arrives in chunks (such as occurs with a file upload), streaming that data through a `WritableStream` object interface provides more flexibility and efficiency.

fs.createWriteStream(path, [options])

The `fs.createWriteStream(path, [options])` method returns a writable stream object for file at `path`.

The following options are available:

- **flags**: File mode argument as a string. Defaults to `w`.
- **encoding**: One of `utf8`, `ascii`, or `base64`. Default to no encoding.
- **mode**: Octal representation of file mode, defaulting to 0666.
- **start**: An offset, indicating the position in the file where writing should begin.

For example, this little program functions as the world's simplest word processor, writing all terminal input to a file, until the terminal is closed:

```
var writer = fs.createWriteStream("novel.txt", 'w');
process.stdin.pipe(writer);
```

Caveats

The side effects of opening a file descriptor and reading from it are minimal, such that in normal development very little thought is given to what is actually happening within the system. Normally, reading a file doesn't change data shape, volume, or availability.

When writing to a file a number of concerns must be addressed, such as:

- Is there sufficient writable storage space available?
- Is another process simultaneously accessing this file, or even erasing it?
- What must be done if a write operation fails or is unnaturally terminated mid-stream?

We've seen the exclusive write mode flag (`wx`) that can help in the case of multiple write processes simultaneously trying to create a file. Full solutions to all the concerns one might face when writing to files are difficult to derive in general, or state briefly. Node encourages asynchronous programming. Nevertheless, with regards the filesystem in particular, sometimes synchronous, deterministic programming is necessary. You are encouraged to keep these and other issues in mind, and to keep I/O non-blocking whenever possible.

Serving static files

Anyone using Node to create a web server will need to respond intelligently to HTTP requests. A HTTP request to a web server for a resource expects some sort of response. A basic file static file server might look like this:

```
http.createServer(function(request, response) {
  if(request.method !== "GET") {
    return response.end("Simple File Server only does GET");
  }
  fs
  .createReadStream(__dirname + request.url)
  .pipe(response);
}).listen(8000);
```

This server services GET requests on port 8000, expecting to find a local file at a relative path equivalent to the URL path segment. We see how easy Node makes it for us to stream local file data, simply piping a `ReadableStream` into a `WritableStream` representing a client socket connection. This is an enormous amount of functionality to be safely implemented in a handful of lines.

Eventually a great deal more would be added, such as handling routines for standard HTTP methods, handling errors and malformed requests, setting proper headers, managing favicon requests, and so forth.

Let's build a reasonably useful file server with Node, one that will respond to HTTP requests by streaming back a resource and which will respect caching requests. In the process we will touch on how to manage content redirection. Later on in this chapter we will also look at implementing file uploads. Note that a web server fully compliant with all features of HTTP is a complicated beast, so what we are creating should be considered a good start, not an end.

Redirecting requests

Sometimes a client will try to `GET` a URL that is incorrect or incomplete in some way, the resource may have been moved, or there are better ways to make the same request. Other times, a `POST` may create a new resource at a new location the client cannot know, necessitating some response header information pointing to the newly created URI. Let's look into two common redirection scenarios someone implementing a static file server with Node might face.

Two response headers are fundamental to redirection:

- `Location`: This indicates a redirection to a location where said content body can be found
- `Content-Location`: This is meant to indicate the URL where the requester will find the original location of the entity enclosed in the response body

As well, there are two specific use cases for these headers:

- To provide information about the location of a newly created resource in response to a `POST`.
- To inform the client of an alternate location for the requested resource in response to a `GET`.

There are many possible pairings of `Location` and `Content-Location` headers with HTTP status codes, the **3xx** (redirection) set in particular. In fact, these headers may even appear together in the same response. The user is encouraged to read the relevant sections of the HTTP/1.1 specification, as only a small set of common cases is discussed here.

Location

Responding to a `POST` with a `201` status code indicates a new resource has been created its URI assigned to the `Location` header, and that the client may go ahead and use that URI in the future. Note that it is up to the client to decide whether, and when, to fetch this resource. As such this is not, strictly speaking, a redirect.

For example a system might create new accounts by posting new user information to a server, expecting to receive the location of a new user page:

```
POST /path/addUser HTTP/1.1
Content-Type: application/x-www-form-urlencoded
name=John&group=friends
…
Status: 201
Location: http://website.com/users/john.html
```

Similarly, in cases where a resource creation request has been accepted but not yet fulfilled, a server would indicate a status of `202`. This would be the case in the preceding example if creation of the new user record had been delegated to a worker queue, which might at some point in the future create a record at the given `Location`.

We will see a realistic implementation demonstrating this usage later on in the chapter, when we discuss file uploads.

Content-Location

When a `GET` is made to a resource that has multiple representations, and those can be found at distinct resource locations, a `content-location` header for the particular entity should be returned.

For example, content format negotiation is a good candidate for `Content-Location` handling. One might be interested in retrieving all blog posts for a given month, perhaps available at a URL such as `http://example.com/september/`. GET requests with an `Accept` header of `application/json` will receive a response in JSON format. A request for XML will receive that representation.

If a caching mechanism is being used those resources may have alternate permanent locations, such as `http://example.com/cache/september.json` or `http://example.com/cache/september.xml`. One would send this additional location information via `Content-Location`, in a response object resembling:

```
Status: 200
Content-Type: application/json
Content-Location: http://blogs.com/cache/allArticles.json
… JSON entity body
```

In cases where the requested URL has been moved, permanently or temporarily, the **3xx** group of status codes can be used with `Content-Location` to indicate this state. For example, to redirect a request to a URL which has been permanently moved one should send a 301 code:

```
function requestHandler(request,response) {
  var newPath = "/thedroids.html";
  response.writeHead(301, {
    'Content-Location': newPath
  });
    response.end();
}
```

Implementing resource caching

A general rule, never expend resources delivering irrelevant information to clients. For a HTTP server, resending files that the client already possesses is an unnecessary I/O cost, exactly the wrong way to implement a Node server, increasing latency as well as the financial hit of paying for misappropriated bandwidth.

Browsers maintain a cache of the files they have already fetched, and an **ETag** (**Entity Tag**) identifies these files. An ETag is a response header sent by servers to uniquely identify entities they are returning, such as a file. When a file changes on a server, that server will send a different ETag for said file, allowing file changes to be tracked by clients.

When a client makes a request to a server for a resource contained within that client's cache, that request will contain an If-None-Match header set to the value of the ETag associated with said cached resource. The If-None-Match header can contain one or multiple ETags:

```
If-None-Match : "686897696a7c876b7e"
If-None-Match : "686897696a7c876b7e", "923892329b4c796e2e"
```

A server understands this header as return the full entity body of the requested resource only if none of the sent ETags match the current resource entity tag. If one of the sent ETags does match the current entity tag, the server will respond with a 304 (not modified) status, which should result in a browser fetching the resource from its internal cache.

Assuming that we have an `fs.Stats` object available, managing cache controls on a resource can be done easily with Node:

```
var etag = crypto.createHash('md5').update(stat.size + stat.mtime).
digest('hex');
if(request.headers['if-none-match'] === etag) {
  response.statusCode = 304;
  return response.end();
} else {
  // stream the requested resource
}
```

We create an `etag` for the current file by creating an MD5 of the current file size and its last modified time, and match against the sent If-None-Match header. If the two do not match the resource representation has changed and the new version must be sent back to the requesting client. Note that the specific algorithm one should use to create an `etag` is not formally specified. The example technique should work well for most purposes.

Hey! What about `Last-Modified` and `If-Unmodified-Since`? These are fine headers, and are also useful in the case of caching files. Indeed, one should set the `Last-Modified` header where possible when responding to entity requests. The techniques we're describing here using ETag would work similarly with these tags, and in fact using both Etags and these other tags is encouraged. For more information, consult: `http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13.3.4`.

Handling file uploads

It is likely that anyone reading this sentence has had at least one experience with uploading a file from a client to a server. Some may have even implemented a file upload service, a server that will receive and do something useful with a multipart data stream. Within popular development environments this task has been made very easy. In the PHP environment, for example, uploaded data is automatically processed and made globally available, neatly parsed and packaged into an array of files or form field values, without the developer having written a single line of code.

Unfortunately, Node leaves implementation of file upload handling to the developer, a challenging bit of work many developers may be unable to successfully or safely complete.

Fortunately, Felix Geisendorfer created the **Formidable** module, one of the most important early contributions to the Node project. A widely implemented, enterprise-grade module with extensive test coverage, it not only makes handling file uploads a snap, but can be used as a complete tool for handling form submissions. We will use this library to add file upload capability to our file server.

For more information about how HTTP file uploads are designed, and the tricky implementation problems developers must overcome, consult the multipart/form-data specification (`http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.2`) and Geisendorfer's breakdown of how **Formidable** was conceived of and evolved (`http://debuggable.com/posts/parsing-file-uploads-at-500-mb-s-with-node-js:4c03862e-351c-4faa-bb67-4365cbdd56cb`).

First, install `formidable` via npm:

```
npm install formidable
```

You can now `require` it:

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

We will assume that file uploads will be posted to our server along a path of `/uploads/`, and that the upload arrives via a HTML form that looks like this:

```
<form action="/uploads" enctype="multipart/form-data" method="post">
Title: <input type="text" name="title"><br />
<input type="file" name="upload" multiple="multiple"><br />
<input type="submit" value="Upload">
</form>
```

This form will allow a client to write some sort of title for the upload, and to select one (or multiple) files for uploading. At this point our only responsibility on our server is to properly detect when a `POST` request has been made and pass the relevant request object to Formidable.

We won't be covering every part of the comprehensive formidable API design, but focusing on the key `POST` events the library exposes. As formidable extends `EventEmitter`, we use the `on(eventName, callback)` format to catch file data, field data, and termination events, sending a response to the client describing what the server has successfully processed:

```
http.createServer(function(request, response) {
  var rm = request.method.toLowerCase();
  if(request.url === '/uploads' && rm === 'post') {
    var form = new formidable.IncomingForm();
    form.uploadDir = process.cwd();
    var resp = "";
    form
    .on("file", function(field, File) {
      resp += "File : " + File.name + "<br />";
    })
    .on("field", function(field, value) {
      resp += field + " : " + value + "<br />";
    })
    .on("end", function() {
      response.writeHead(200, {'content-type': 'text/html'});
      response.end(resp);
    })
    .parse(request);
    return;
  }
}).listen(8000);
```

We see here how a `formidable` instance receives an `http.Incoming` object through its `parse` method, and how the write path for incoming files is set using the `uploadDir` attribute of that instance. The example sets this directory to the local directory. A real implementation would likely target a dedicated upload folder, or even direct the received file to a storage service, receiving in return the final storage location (perhaps receiving it via HTTP and a `Location` header…).

Note as well how the file event callback receives a formidable `File` object as a second argument, which contains important file information including:

- **size**: The size of the uploaded file, in bytes
- **path**: The current location of the uploaded file on the local filesystem, such as `/tmp/bdf746a445577332e38be7cde3a98fb3`

- **name**: The original name of the file as it existed on the client filesystem, such as `lolcats.jpg`
- **type**: The file mime type, such as `image/png`

In a few lines of code we've implemented a significant amount of `POST` data management. Formidable also provides tools for handling progress indicators, dealing with network errors, and more, which the reader can learn about by visiting `https://github.com/felixge/node-formidable`.

Putting it all together

Recalling our discussion of favicon handling from the previous chapter, and adding what we've learned about file caching and file uploading, we can now construct a simple file server handling `GET` and `POST` requests:

```
http.createServer(function(request, response) {
  var rm = request.method.toLowerCase();
  if(rm === "post") {
    var form = new formidable.IncomingForm();
    form.uploadDir = process.cwd();
    form
    .on("file", function(field, file) {
      // process files
    })
    .on("field", function(field, value) {
      // process POSTED field data
    })
    .on("end", function() {
      response.end("Received");
    })
    .parse(request);
    return;
  }
  // We can only handle GET requests at this point
  if(rm !== "get") {
    return response.end("Unsupported Method");
  }
  var filename = __dirname + request.url;
  fs.stat(filename, function(err, stat) {
      if(err) {
        response.statusCode = err.errno === 34 ? 404 : 500;
      return response.end()
      }
```

```
    var etag = crypto.createHash('md5').update(stat.size + stat.
mtime).digest('hex');
    response.setHeader('Last-Modified', stat.mtime);
    if(request.headers['if-none-match'] === etag) {
      response.statusCode = 304;
      return response.end();
    }

    response.setHeader('Content-Length', stat.size);
    response.setHeader('ETag', etag);
    response.statusCode = 200;
    fs.createReadStream(filename).pipe(response);
  });
}).listen(8000);
```

Note the 404 (not found) and 500 (internal server error) status codes.

`Content-Length` is measured in bytes, not characters. Normally your data will be in single byte characters (hello is five bytes long), but this is not always the case. If you are determining the length of a stream buffer, use `Buffer.byteLength`.

Summary

In this chapter we've seen how Node's API is a comprehensive map to native filesystem bindings, exposing a full range of functionality to the developer while requiring very little code or complexity. Additionally, we've seen how files are easily wrapped into `Stream` objects, and how this consistency with the rest of Node's design simplifies interactions between different types of I/O, such as between network data and files.

We've also learned something about how to build servers with Node that can accommodate regular client expectations, easily implementing file uploading and resource caching. Having covered the key features of Node, it is time to use these techniques in building larger applications able to handle many thousands of clients.

5
Managing Many Simultaneous Client Connections

If everyone helps to hold up the sky, then one person does not become tired.

— *Tshi Proverb*

Maintaining a high level of throughput while managing thousands of simultaneous client transactions in the unpredictable and "bursty" environment of networked software is one expectation developers have for their Node implementations. Given a history of failed and unpopular solutions, the problem of concurrency has even been assigned its own numeronym: "The C10K problem". How should network software confidently serving 10,000 simultaneous clients be designed?

The question of how to best build high concurrency systems has provoked much theory over the last several decades, with the debate mostly between two alternatives, **threads** and **events**:

> *Threading allows programmers to write straight-line code and rely on the operating system to overlap computation and I/O by transparently switching across threads. The alternative, events, allows programmers to manage concurrency explicitly by structuring code as a single-threaded handler that reacts to events (such as non-blocking I/O completions, application-specific messages, or timer events).*
>
> — "*A Design Framework for Highly Concurrent Systems*" *(Welsh, Gribble, Brewer & Culler, 2000), p. 2.* `http://www.eecs.harvard.edu/~mdw/papers/events.pdf`

Two important points are made in the preceding quote:

- Developers prefer to write *structured* code (straight-line; single-threaded) that hides the complexity of multiple simultaneous operations where possible
- I/O efficiency is a primary consideration of high-concurrency applications

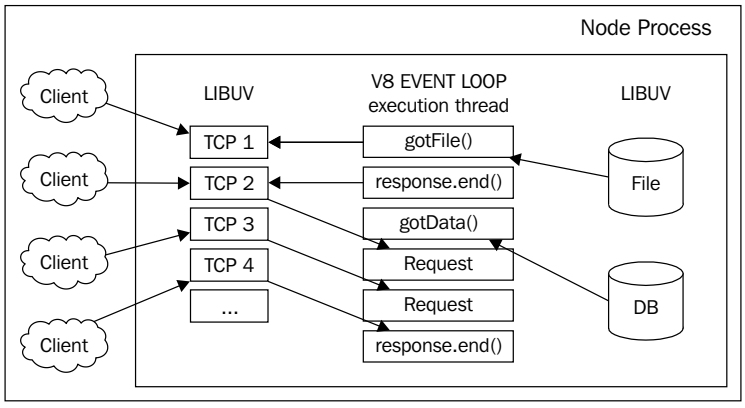
Until very recently, programming languages and related frameworks were not (necessarily) optimized for software executing across nodes in a distributed network, or even across processors. Algorithms are expected to be deterministic; data written to a database is expected to be immediately available for reading. In this age of eventually consistent databases and asynchronous control flow, developers can no longer expect to know the precise state of an application at any given point of time a sometimes mind-bending challenge for the architects of highly concurrent systems.

As we learned in *Chapter 2, Understanding Asynchronous Event-Driven Programming*, Node's design attempts to combine the advantages of both threads and events, serving all clients on a single thread (an event loop wrapping a JavaScript runtime) while delegating the blocking work (I/O) to an optimized thread pool that informs the main thread of state changes via an event notification system.

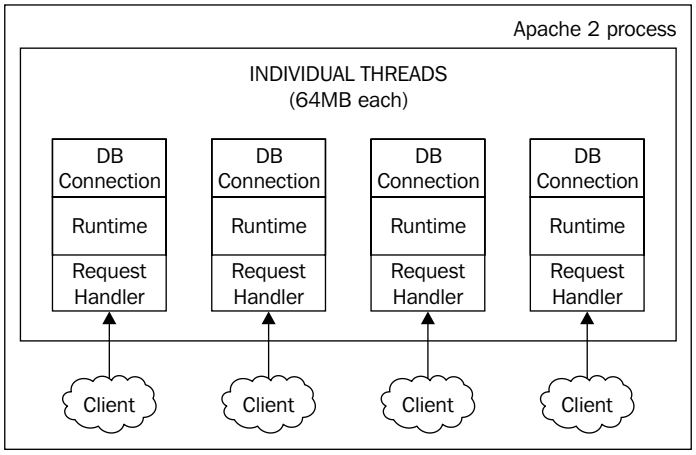
Think clearly about how the following HTTP server implementation, running on a single CPU, is responding to each client request by wrapping a callback function in the context of the request and pushing that execution context onto a stack that is constantly emptied and rebuilt within a single thread bound to an event loop:

```
require('http').createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello client from' + req.connection.remoteAddress);
  console.log(req);
}).listen(8000);
```

Schematically:



On the other hand, a server like Apache spins up a thread for each client request:



These two approaches are very different. The claim implicit in Node's design is this: it is easier to reason about highly concurrent software when program flow is organized along a single thread, and that decreasing I/O latency increases the number of simultaneous clients that can be supported even in a single-threaded execution model. The second claim will be tested later, but for now let's see how easy it can be to build basic processes that naturally scale.

We will demonstrate how to track and manage the relationships between concurrent processes using Node, in particular, those servicing multiple clients simultaneously. Our goal is to set up a basic understanding of how **state** should be modeled within a Node server or other processes. How is it that a large online social network serves you customized information tailored by your friendships or interests? How is your shopping cart maintained over several shopping sessions, without disappearing, even containing suggestions based on your history of purchases? How can a single client interact with other clients?

Understanding concurrency

We would all agree that there are unexpected events in the world, and that many of them occur at exactly the same time. It is also clear that the state of any given system may be composed of any number of sub-states, where the full consequence of even minor state changes are difficult to predict—the power of a butterfly's wings being enough to tip a much larger system into an alternate state. And we also know that the volume and shape of a system, over time, changes in ways difficult to predict.

In his PHD thesis "Foundations of Actor Semantics", written in 1981, *William Clinger* proposed that his work was:

> *...motivated by the prospect of highly parallel computing machines consisting of dozens, hundreds or even thousands of independent microprocessors, each with its own local memory and communications processor, communicating via a high-performance communications network.*

As it turns out, Clinger was on to something. Concurrency is a property of systems composed of many simultaneously executing operations, and the network software we are now building resembles the one he envisioned, only much larger, where "hundreds or even thousands" is the lower bound, not the higher.

Node makes concurrency accessible, while simultaneously scaling across multiple cores, multiple processes, and multiple machines. It is important to note that Node places as much importance on the simplicity and consistency of programs as it does on being *the fastest* solution, embracing and enforcing non-blocking I/O in an effort to deliver high concurrency through well-designed and predictable interfaces. This is what Dahl meant when he said, "Node's goal is to provide an *easy* way to build *scalable* network programs".

Happily, it also turns out that Node is very fast.

Concurrency is not parallelism

A problem can be solved by dividing some of it into smaller problems, spreading those smaller problems across a pool of available people or workers to work on in parallel, and delivering the parallel results concurrently.

Multiple processes each solving one part of a single mathematical problem simultaneously is an example of parallelism.

Rob Pike, general wizard hacker and co-inventor of Google's Go programming language defines concurrency in this way:

> *Concurrency is a way to structure a thing so that you can, maybe, use parallelism to do a better job. But parallelism is not the goal of concurrency; concurrency's goal is a good structure.*

Successful high-concurrency application development frameworks provide a simple and expressive vocabulary for describing such systems.

Node's design suggests that achieving its primary goal—to provide an easy way to build scalable network programs—includes simplifying how the execution order of coexisting processes is structured and composed. Node helps a developer struggling with a program within which many things are happening at once (such as serving many concurrent clients) to better organize his or her code.

This is not to say that Node is designed to concede efficiency in order to maintain simple interfaces—far from it. Instead, the idea is to move responsibility for implementing efficient parallel processing away from the developer and into the core design of the system, leaving the developer free to structure concurrency through a simple and predictable callback system, safe from deadlocks and other traps.

Node's bracing simplicity comes at a good time, as social and community networks grow alongside the world's data. Systems are being scaled to sizes that few would have predicted. It is a good time for new thinkings, such as how to describe and design these systems, and the way they make requests of, and respond to, each other.

Routing requests

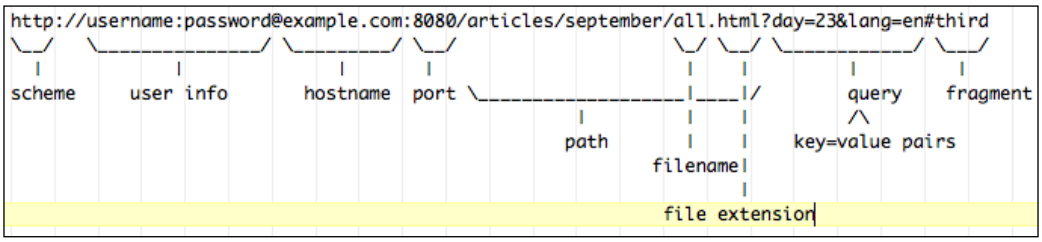
HTTP is a data transfer protocol built upon a request/response model. Using this protocol many of us communicate our current status to friends, buy presents for family, or discuss a project over e-mail with colleagues. A staggering number of people have come to depend on this foundational Internet protocol.

Typically, a browser client will issue an HTTP GET request to a server. This server then returning the requested resource, often represented as an HTML document. HTTP is stateless, which simply means that each request or response maintains no information on previous requests or responses—with each back and forward movement through web pages the entire browser state is destroyed and rebuilt from scratch.

Servers route state change requests from clients, ultimately causing new state representations to be returned, which clients (often browsers) redraw or report. When the WWW was first conceived this model made sense. For the most part this new network was understood as a distributed filesystem, accessible by anyone with a web browser, where a specific resource (such as a newspaper article) could be requested from a file-serving computer (a server) located somewhere on the network (at an **Internet Protocol** or **IP** address) via an HTTP request (such as GET) by simply typing in a URL (for example, `www.example.org/articles/april/showers.html`). A user requests a page and that page appears, perhaps containing (hyper) links to related pages.

However, since a stateless protocol does not maintain context information it was nearly impossible for the operator of a server to develop a more interesting relationship with a visitor across a series of requests, or for a visitor to dynamically aggregate multiple responses into a single view.

Additionally, the expressiveness of requests was limited both by the protocol itself and by the lack of server content rich enough to usefully support a more descriptive vocabulary. For the most part, requests were as blunt as pointing a finger at an object of desire—"Get that for me". Consider the parts of a typical URL:



We can see how much client work is involved in describing a simple resource location, with query parameters and resource targets an awkward afterthought, becoming nearly unusable once more than a few resource descriptors are used. While this was workable in a time of simple documents in well-understood and invariant hierarchies, the demands and complexities of modern networked software have rendered the original concept unworkable and in need of improvement.

The clumsiness of passing around increasingly complex key/value pairs to maintain user state began to frustrate the ambitions of this new medium. Very quickly it became obvious to developers that a growing reliance on the Internet as the utility communication layer undergirding the world's information, software, and commerce required a more refined approach.

Over time these limitations have been overcome through a combination of improvements to the HTTP protocol, the introduction of JavaScript to the browser, technologies such as browser cookies and the attendant innovation from developers building products and services exploiting these advances.

Nevertheless, the HTTP protocol itself continues to be dominated by the same motifs of individual file-like resources existing at a distinct and permanent path and identified by an often nondescriptive name.

What actually exists on many servers now is a complex software specifying network interfaces to data models. Communicating with these types of networked applications involves getting and setting the state of that data model, both in general and as it applies in particular to the client making a request.

Clients deploying a real-time solution both set and get resource state representations on a server. Application servers must report a client's state in relation to multiple processes (databases, files, rules engines, calculation engines, and so on) on each request, and often unilaterally on application state changes (for example, a user losing access permission). Clients are often not browsers, but other servers. How should they communicate?

Understanding routes

Routes map URLs to actions. Rather than constructing an application interface in terms of URL paths to specific files that contain some logic, designing with routes involves assigning a specific function to a distinct combination of a URL path and request method. For example, a web service that accepts requests for lists of cities might be called in this manner:

```
GET /services/cities.php?country=usa&state=ohio
```

When your server receives this request it would pass the URL information to a PHP process that will execute the application logic in `cities.php`, such as reading the query, parsing out the country and state, calling a database, building a response, and returning it.

Node has the great benefit of being able to function both as the server and the application environment. The server can field requests directly. It makes more sense, then, to use URLs as simple statements of intent:

```
GET /listCities/usa/ohio
```

In a Node server we might use something like the following code to handle these requests for cities:

```
var app = http.createServer(function(request, response) {
  var url     = request.url;
  var method     = request.method;
  if(method === "GET") {
      if(url === "/listCities/usa/ohio") {
          database.call("usa","ohio",function(err, data) {
              response.writeHead(200, {'Content-Type':
                'application/json' });
            // Return list of cities in Ohio, USA
              response.end(JSON.stringify(data));
          });
      }
      if(url === "/listCities/usa/arizona") { ... }
      if(url === "/listCities/canada/ontario") { ... }
  }
```

One good thing and one bad jump out:

- URL handling is clearly organized in one place
- The code is impossibly repetitive

Writing out every possible route won't work. We'll keep the organization, but need to create variables in routes, preferring to define a general route expression like this:

```
/listCities/:country/:state
```

The method `listCities` can accept `country` and `state` *variable* arguments, identified with a colon (`:`) prefix. Within our server we would need to convert this symbolic expression into a regular expression. In this case the `RegExp /^\/listCities\/([^\/\.]+)\/([^\/\.]+)\/?$/` could be used to extract ordered values from our example URL into a value map similar to:

```
{  country: "usa",
    state: "ohio"  }
```

By treating requests as expressions our server design has become a little saner, nicely routing any country/state combination to a common handler function:

```
  if(request.method === "GET") {
    var match = request.url.match(/^\/listCities\/([^\/\.]+)\/
([^\/\.]+)\/?$/);
       if(match) {
          database.call(match[1],match[2],function(err, data) {…}

    }
```

This form of request routing has "won the argument" in the Node community, being the default behavior of various frameworks and tools. In fact, this way of thinking about routing requests has gained acceptance in many other development environments, such as Ruby on Rails. As a result, most web application frameworks for Node have been developed around routing.

The most popular web application framework for Node is *T.J. Holowaychuk*'s **Express** framework, and we'll be using this framework frequently in this book when designing routing servers. You can install it via `npm` by running `npm install -g express`.

Using Express to route requests

Express simplifies the complexity of defining route-matching routines. Our example might be written in the following way using Express:

```
var express = require('express');
var app = express();
app.get('/listCities/:country/:state', function(request, response){
  var country = request.params.country;
  var state = request.params.state;
  response.end("You asked for country: " + country + " and state: " +
state);
});
app.listen(8080);

GET /listCities/usa/ohio
// You asked for country: usa and state: ohio
GET /didnt/define/this
// Cannot GET /didnt/define/this
GET /listCities // note missing arguments
// Cannot GET /listCities
```

Instantiating Express delivers a fully-formed web server wrapped in an easy-to-use application development API. Our cities service is clearly defined and its variables stated, expecting to be called via GET (one might also use `app.post(...)` or `app.put(...)` or any other standard HTTP method).

Express also introduces the idea of chaining request-handling routines, which in Express are understood as **middleware**. In our example we are calling a single function in order to handle a cities request. What if, prior to calling our database, we want to check that the user is authenticated? We might add an `authenticate()` method *prior* to our main service method:

```
var authenticate = function(request, response, next) {
    if(validUser) { next(); }
    else { response.end("INVALID USER!"); }
    }
app.get('/listCities/:country/:state', authenticate,
function(request, response) { ... });
```

Middleware can be chained, in other words, simplifying the creation of complex execution chains, nicely following the **Rule of Modularity**. Many types of middleware for handling favicons, logging, uploads, static file requests, and so on have already been developed. To learn more, visit the following link:

`http://www.express.com`

Having established the proper way for Node servers to be configured for routing requests, we can now begin a discussion on how to identify the client making the request, assigning that client a unique session ID, and managing that session through time.

Using Redis for tracking client state

For some of the applications and examples in this chapter we will be using **Redis**, an in-memory **Key/Value** (**KV**) database developed by *Salvatore Sanfilippo* and currently supported by Pivotal. More information on Redis can be found at `http://redis.io`. A well-known competitor to Redis is **Memcached** (`http://memcached.org`).

In general, any server that must maintain the session state of many clients will need a high speed data layer with near-instantaneous read/write performance, as request validation and user state transformations can occur multiple times on each request. Traditional file-backed relational databases tend to be slower at this task than in-memory KV databases. We're going to use Redis for tracking the client state.

Redis is a single-threaded server with a straightforward install:

```
wget http://download.redis.io/redis-stable.tar.gz
tar xvzf redis-stable.tar.gz
cd redis-stable
make
```

There is now a server and a command-line utility available in the `/src` folder of the installation folder. It's a good idea to make these easily accessible on the command line:

```
sudo cp redis-server /usr/local/bin/
sudo cp redis-cli /usr/local/bin/
```

To start Redis, run the following command line:

```
redis-server
```

To interact with Redis:

```
redis-cli
```

The preceding steps are sufficient when exploring Redis development. In production, one will need to properly configure the build, allocating memory, setting passwords, and so on. When ready, you can seek further information from the following link:

`http://redis.io/topics/quickstart`

Notably, Amazon's **ElastiCache** service enables Redis "in the cloud" as an in-memory cache, with automatic scaling and redundancy:

`http://aws.amazon.com/elasticache/`

Redis supports a standard interface for expected actions, such as getting or setting Key/Value pairs. To `get` the value stored at a key, first start the Redis CLI:

```
redis-cli
redis> get somerandomkey
(nil)
```

Redis will return (nil) when a key does not exist. Let's `set` a key:

```
redis> set somerandomkey "who am I?"
redis> get somerandomkey
"who am I?"
```

To use Redis within a Node environment we will need some sort of binding. We will be using *Matt Ranney*'s `node_redis` module. Install it with npm using the following command line:

```
npm install hiredis redis
```

To set a value in Redis and get it back again we can now do this in Node:

```
var redis     = require("redis");
var client    = redis.createClient();
client.set("userId", "jack", function(err) {
    client.get("userId", function(err, data) {
        console.log(data); // "jack"
    });
});
```

Storing user data

Managing many users means at least tracking their user information, some stored long term (for example, address, purchase history, and contact list) and some session data stored for a short time (time since login, last game score, and most recent answer).

Normally we would create a secure interface or similar, allowing administrators to create user accounts. It will be clear to the reader how to create such an interface by the end of this chapter. For the examples that follow, we'll only need to create one user, to act as a volunteer. Let's create `Jack`:

```
redis> hset jack password "beanstalk"
redis> hset jack fullname "Jack Spratt"
```

This will create a key in Redis—`Jack`—containing a hash resembling:

```
{
    "password": "beanstalk",
    "fullname": "Jack Spratt"
}
```

If we wanted to create a hash and add several KV pairs all at once, we could achieve the preceding with the `hmset` command:

```
redis> hmset jack password "beanstalk" fullname "Jack Spratt"
```

Now `Jack` exists:

```
redis> hgetall jack
1) "password"
2) "beanstalk"
3) "fullname"
4) "Jack Spratt"
```

We can use the following command to fetch the value stored for a specific field in `Jack`'s account:

```
redis> hget jack password // "beanstalk"
```

Handling sessions

How does a server know if the current client request is part of a chain of previous requests? Web applications engage with clients through long transactional chains—the shopping cart containing items to buy will still be there even if a shopper navigates away to do some comparison-shopping. We will call this a **session**, which may contain any number of KV pairs, such as a username, product list, or the user's login history.

How are sessions started, ended, and tracked? There are many ways to attack this problem, depending on many factors existing in different ways on different architectures. In particular, if more than one server is being used to handle clients, how is session data shared between them?

We will use cookies to store session IDs for clients, while building a simple long-polling server. Keep in mind that as applications grow in complexity this simple system will need to be extended. As well, long-polling as a technology is giving ground to the more powerful socket techniques we will explore in our discussions around building real-time systems. However, the key issues faced when holding many connected clients simultaneously on a server, and tracking their sessions, should be demonstrated.

Cookies and client state

Netscape provided the preliminary specification for cookies, in 1997:

> *Cookies are a general mechanism which server side connections...can use to both store and retrieve information on the client side of the connection. The addition of a simple, persistent, client-side state significantly extends the capabilities of Web-based client/server applications...*
>
> *A server, when returning an HTTP object to a client, may also send a piece of state information which the client will store... Any future HTTP requests made by the client...will include a transmittal of the current value of the state object from the client back to the server. The state object is called a cookie, for no compelling reason.*
>
> — *http://lib.ru/WEBMASTER/cookie_spec.txt_with-big-pictures.html*

Here we have one of the first attempts to "fix" the stateless nature of HTTP, specifically the maintenance of session state. It was such a good attempt, which still remains a fundamental part of the Web.

We've already seen how to read and set the `Cookie` header with Node. Express makes the process a little easier:

```
var express = require('express');
var app = express();
app.use(express.cookieParser());
app.get('/mycookie', function(request, response){
        response.end(request.cookies.node_cookie);
});
app.get('/', function(request, response) {
    response.cookie('node_cookie', Math.floor(Math.random() * 10e10));
    response.end("Cookie set");
});
app.listen(8000);
```

Note the `use` method, which allows us to turn on the cookie handling middleware for Express. Here we see that whenever a client hits our server this client is assigned a random number as a cookie. By navigating to `/mycookie` this client can see the cookie.

A simple poll

We need to create a concurrent environment, one with many simultaneously connected clients. We'll use a long-polling server to do this, broadcasting to all connected clients via `stdin`. Additionally, each client will be assigned a unique session ID, used to identify the client's `http.serverResponse` object, which we will push data to.

Long polling is a technique whereby a server holds on to a client connection until there is data available to send. When data is ultimately sent to the client, the client reconnects to the server and the process continues. It was designed as an improvement on **short polling**, which is the inefficient technique of blindly checking with a server for new information every few seconds or so, hoping for new data. Long polling only requires a reconnection following a tangible delivery of data to the client.

We'll use two routes. The first route is described using a forward slash (`/`), a root domain request. Calls to this path will return some HTML forming the client UI. The second route is `/poll`, which the client will use to reconnect with the server following the receipt of some data.

The client UI is extremely simple: its sole purpose is to make an **XML HTTP request** (**XHR**) to a server (which will hold that request until some data arrives), repeating this step immediately following the receipt of some data. Our UI will display a list of messages received within an unordered list. For the XHR bit we will use the jQuery library. Any similar library can be used, and building a pure JavaScript implementation is not difficult.

HTML:

```
<ul id="results"></ul>
```

JavaScript:

```
function longPoll() {
    $.get('http://www.pathjs.com/poll', function (data) {
        $('<li>' + data + '</li>').appendTo('#results');
        longPoll();
    });
}
longPoll();
```

The server is also simple, mainly responsible for setting session IDs and holding on to concurrent client connections until such time as data is available, which is broadcast to all connected clients. These connections are indexed via session IDs, maintained using cookies:

```
var connections = {};
app.use(express.cookieParser());
app.get('/poll', function(request, response){
    var id = request.cookies.node_poll_id;
    if(!id) {
        return;
    }
    connections[id] = response;
});
app.get('/', function(request, response) {
    fs.readFile('./poll_client.html', function(err, data) {
        response.cookie('node_poll_id', Math.floor(Math.random() *
10e10));
        response.writeHead(200, {'Content-Type': 'text/html'});
        response.end(data);
    });
});
app.listen(2112);
```

Once a client has fetched the UI it is ready to receive data. In this example we will allow data arriving via `stdin` to be broadcast to all connected clients:

```
var broadcast = function(msg) {
    var conn;
    for(conn in connections) {
        connections[conn].end(msg);
    }
}
process.stdin.on('readable', function() {
    var msg = this.read();
    msg && broadcast(msg.toString());
});
```

Run this server on the command line, and connect to the server via a browser. A nearly blank page should be displayed. Return to the command line and enter some text—this message should immediately appear in your browser. Try it with several open browser windows, which should all receive what you've entered.

This small example demonstrates another advantage of Node's evented single-threaded model, especially when it comes to building servers, closure. We see how a single function, receiving request/response bindings for a unique client can, via closure, access common data safely. Here the `broadcast` method is accessing members of the `connections` object created earlier within `app.get`—one method scope is accessing data created within an external scope. Each call to the HTTP request handler is protected from interference from another thread or process, yet can be exposed to useful shared data via closure.

An interesting use of the new **WeakMap** feature of JavaScript is using a request object as a key in a map (remember to use the harmony flag when starting your Node process).

Centralizing states

By keeping user sessions centralized in a Redis database we are able to have multiple servers access the same session data. This is the primary advantage of centralizing session data. Additionally, analytics on current sessions (and even past sessions) can be done.

We've seen how we might use cookies to track user connections. The primary problem with this solution is that it only works on a single server. What if our system is running two separate client servers? The connection map we used earlier is determined by connections to the *local* server. What if data is written from a *remote* server? How do we ensure that all servers are informed of all events, such that each can push new information to all of *its* clients?

Redis introduces the idea of **pub/sub**, or publish/subscribe, which can be used to deliver messages across many Node processes. Usage is simple and intuitive:

```
var redis     = require("redis");
var receiver    = redis.createClient();
var publisher     = redis.createClient();
receiver.subscribe("firehose");
receiver.on("message", function(channel, message) {
    console.log("Received message: " + message + " on channel: " +
      channel);
});
setTimeout(function() {
    publisher.publish("firehose", "Hello!");
}, 1000);
```

After creating the `publisher` and `receiver` clients, we wait about one second and publish a message to a channel, which in this case has a subscriber to notify of the message receipt.

The problem with scaling our simple polling infrastructure is solved by making a few small changes. Consider again the code that catches and broadcasts messages:

```
Var broadcast = function(msg) {
    var conn;
    for(conn in connections) {
        connections[conn].end(msg);
    }
}
process.stdin.on('readable', function() {
    var msg = this.read();
    msg && broadcast(msg.toString());
});
```

Let's convert this into a publish/subscribe system, allowing any number of these servers to exist while maintaining full broadcast coverage when new messages arrive:

```
var redis     = require("redis");
var receiver    = redis.createClient();
var publisher     = redis.createClient();

receiver.subscribe("stdin_message");
receiver.on("message", function(channel, message) {
    var conn;
    for(conn in connections) {
        connections[conn].end(message);
    }
```

```
        console.log("Received message: " + message + " on channel: " +
          channel);
    });
    process.stdin.on('readable', function() {
        var msg = this.read();
        msg && publisher.publish("stdin_message", msg.toString());
    });
```

Rather than favoring the local server, all message broadcasting is made "network aware", giving all instances of this server a chance to broadcast to the clients it is managing, regardless of which server is actually receiving the message data.

As we progress through larger applications in later chapters, more advanced network-messaging techniques will be unpacked. One key takeaway is that the replication of Node servers is *the* typical Node scaling strategy. Plan for a distributed session store, and for building an intra-process communication layer to do that distribution.

Authenticating connections

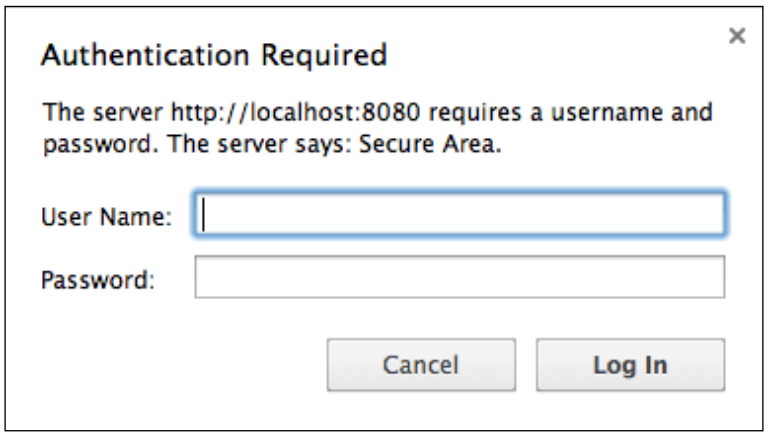
In conjunction with establishing client session objects, a Node server often demands authentication credentials. The theory and practice of web security is extensive. We want to simplify our understanding into two main authentication scenarios:

- When the wire protocol is HTTPS
- When it is HTTP

The first is naturally secure, and the second is not. For the first we will learn how to implement Basic authentication in Node, and for the second a challenge-response system will be described.

Basic authentication

As mentioned, Basic authentication sends plain text over the wire containing a username/password combination, using standard HTTP headers. It is a simple and well-known protocol. Any server sending the correct headers will cause any browser to display a login dialog, like the following one:



Nonetheless this method remains insecure, sending non-encrypted data in plain text over the wire. For the sake of simplicity we will demonstrate this authentication method on a HTTP server, but it must be stressed that in real-world usage the server *must* be communicating via a secure protocol, such as HTTPS.

Let's implement this authentication protocol with Node. Employing the user database developed earlier in Redis, we validate submitted credentials by checking user objects for matching passwords, handling failures and successes.

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

http.createServer(function(req, res) {
    var auth = req.headers['authorization'];
    if(!auth)
       {
```

By sending a `401` status and the `'authorization'` header on a new client connection, a dialog like the one previous screenshot will appear:

```
            res.writeHead(401, {'WWW-Authenticate': 'Basic
              realm="Secure Area"'});
            return res.end('<html><body>Please enter some
              credentials.</body></html>');
        }
        var tmp = auth.split(' ');
```

Split on a space, the original auth string looks like **Basic Y2hhcmxlczoxMjM0NQ==** and we need the second part. Note the `'base64'` transformation in the following code:

```
        var buf = new Buffer(tmp[1], 'base64');
        var plain_auth = buf.toString();
        var creds = plain_auth.split(':');
        var username = creds[0];
        var password = creds[1];
        //    Find this user record
        client.get(username, function(err, data) {
            if(err || !data) {
                res.writeHead(401, {'WWW-Authenticate': 'Basic
                  realm="Secure Area"'});
                return res.end('<html><body>You are not
                  authorized.</body></html>');
            }

            res.statusCode = 200;
            res.end('<html><body>Welcome!</body></html>');
        });
    }).listen(8080);
```

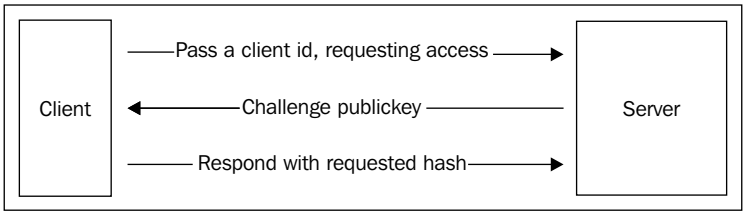
In this way a straightforward login system can be designed. Because browsers will naturally prompt users requesting access to a protected domain, even the login dialog is taken care of.

To learn about implementing Basic authentication using Express, visit the following link:

`http://expressjs.com/api.html#basicAuth`

Handshaking

Another authentication method to consider in situations where an HTTPS connection cannot be established is a challenge-response system:



In this scenario a client asks a server for access for a specific user, username, ID, or similar. Typically this data would be sent via a login form. Let's mock up a challenge/response scenario, using for our example the user we created earlier—`Jack`.

The general design and purpose of a challenge/response system aims to avoid sending any password data in plain text over the wire. So, we will need to decide on an encryption strategy that both the client and the server share. For our example let's use the **SHA256** algorithm. Node's `crypto` library contains all of the tools necessary for creating this type of hash. The client likely does not, so we must provide one. We'll use the one developed by *Chris Veness*, which can be downloaded from the following link:

`http://csrc.nist.gov/groups/ST/toolkit/examples.html`

To initiate this login the client will need to send an authentication request for the user `Jack`:

```
GET /authenticate/jack
```

In response the client should receive a server-generated public key—the challenge. The client must now form a string of `Jack`'s password prefixed by this key. Create a SHA256 hash from it, and pass the resulting hash to `/login/`. The server will have also created the same SHA256 hash—if the two match, the client is authenticated:

```
<script src="sha256.js"></script>
<script>
$.get("/authenticate/jack", function(publicKey) {
    if(publicKey === "no data") {
```

```
            return alert("Cannot log in.");
        }
        //    Expect to receive a challenge: the client should be able
         //      to derive a SHA456 hash
        //    String in this format: publicKey + password. Return that
        //    string.
        //
        var response = Sha256.hash(publicKey + "beanstalk");
        $.get("/login/" + response, function(verdict) {
            if(verdict === "failed") {
                return alert("No Dice! Not logged in.");
            }
            alert("You're in!");
        });
    });
    </script>
```

The server itself is very simple, composed of the two mentioned authentication routes. We can see in the following code how, upon receipt of a username, (`jack`) the server will first check for an existing user hash in Redis, breaking the handshake if no such data is found. If a record exists we create a new, random public key, compose the relevant SHA256 hash, and return this challenge value to the client. Additionally, we set this hash as a key in Redis, with its value being the sent username:

```
var crypto    = require('crypto');
var fs        = require('fs');
var express   = require('express');
var app       = express();
var redis     = require("redis");
var client    = redis.createClient();

app.get('/authenticate/:username', function(request, response){
    var publicKey = Math.random();
    var username = request.params.username; // This is always
      "jack"
    // ... get jack's data from redis
    client.hgetall(username, function(err, data) {
        if(err || !data) {
            return response.end("no data");
        }
        //    Creating the challenge hash
        var challenge =
          crypto.createHash('sha256').update(publicKey +
          data.password).digest('hex');
        //    Store challenge for later match
```

```
            client.set(challenge, username);
            response.end(challenge);
        });
    });
    app.get('/login/:response', function(request, response){
        var hash = request.params.response;
          client.exists(hash, function(err, exists) {
              if(err || !exists) {
                  return response.end("failed");
              }
          });
          client.del(hash, function() {
              response.end("OK");
          });
    });
```

In the `/login/` route handler we can see how a check is made if the response exists as a key in Redis and, if found, we immediately delete the key. This is necessary for several reasons, not least of which is preventing others to send the same response and gain access. We also generally don't want these now useless we want keys to pile up. This presents a problem: *what if a client never responds to the challenge?* As the key cleanup only happens when a `/login/` attempt is made, this key will never be removed.

Unlike most KV data stores Redis introduces the idea of **key expiry**, where a `set` operation can specify a **Time To Live** (**TTL**) for a key. For example, here we use the `setex` command to set a key `userId` to value `183` and specify that this key should expire in one second:

```
client.setex("doomed", 10, "story", function(err) { ... });
```

This feature offers an excellent solution to our problem. By replacing the `client.set(challenge, username);` line with the following line:

```
client.setex(challenge, 5, username)
```

we ensure that, no matter what, this key will disappear in `5` seconds. Doing things this way also functions as a light security measure, leaving a very short window for a response to remain valid, and being naturally suspicious of delayed responses.

Summary

Node provides a set of tools that help in the design and maintenance of large-scale network applications facing the C10K problem. In this chapter we've taken our first steps into creating network applications with many simultaneous clients, tracking their session information and their credentials. This exploration into concurrency has demonstrated some techniques for routing, tracking, and responding to clients. We've touched on some simple techniques to use when scaling, such as the implementation of intra-process messaging via a publish/subscribe system built using a Redis database.

We are now ready to go deeper into the design of real-time software—the logical next step after achieving high concurrency and low latency by using Node. We will extend the ideas outlined during our discussion of long polling and place them in the context of more robust problems and solutions.

Further reading

Concurrency and parallelism are rich concepts that have enjoyed rigorous study and debate. When an application architecture is designed to favor threads, events, or some hybrid, it is likely that the architects are opinionated about both concepts. You are encouraged to dip a toe into the theory and read the following articles. A clear understanding of precisely what the debate is about will provide an objective framework that can be used to qualify a decision to choose (or not choose) Node.

- Some numbers:

  `https://cs.uwaterloo.ca/~brecht/papers/getpaper.php?file=eurosys-2007.pdf`

- Threads are a bad idea:

  `http://www.cs.sfu.ca/~vaughan/teaching/431/papers/ousterhout-threads-usenix96.pdf`

- Events are a bad idea:

  `http://static.usenix.org/events/hotos03/tech/full_papers/vonbehren/vonbehren.pdf`

- How about together?

  `http://repository.upenn.edu/cgi/viewcontent.cgi?article=1391&context=cis_papers`

- It's a false dichotomy:

  `http://swtch.com/~rsc/talks/threads07/`

6
Creating Real-time Applications

Nothing endures but change.

—*Heraclitus*

What is real-time software? A list of friends gets updated the instant one joins or exits. Traffic updates automatically stream into the smartphones of drivers looking for the best route home. The sports page of an online newspaper immediately updates scoreboards and standings as points are scored in an actual game. Users of this type of software expect reactions to change to be communicated quickly, and this expectation demands a particular focus on reducing network latency from the software designer. Data I/O updates *must* occur along subsecond time frames.

Let's step back and consider the general characteristics of the Node environment and community that make it an excellent tool for creating these kinds of responsive network applications.

Some validation of Node's design, it may be argued, is found in the enormous community of open developers contributing enterprise-grade Node systems, some teams financially supported and fully backed by Internet companies as large as Microsoft, LinkedIn, eBay and others. Multicore, multiserver enterprise systems are being created using free software mostly written in JavaScript.

Why are so many companies migrating toward Node when designing or updating their products? The following list enumerates the reasons why:

- Node offers the excellent npm package management system, which integrates easily with the Git version control system. A shallow learning curve helps even inexperienced developers safely store, modify, and distribute new modules, programs, and ideas. Developers can develop private modules on private Git repositories and distribute these repositories securely within a private network using npm. As a result, the community of Node users and developers has rapidly expanded, some members gaining great fame. *If you build it, they will come*.
- Node lifted the system-access barrier for a large group of skilled programmers, suddenly releasing pent-up talent into an empty volume, offering the ecosystem of opportunity that a popular new project in need of many improvements in infrastructure brings. The point is this: Node merged the opportunity of concurrency with native JavaScript events; its brilliantly designed API allowed users of a well-known programming paradigm to take advantage of high-concurrency I/O. *If you reward them, they will come*.
- Node lifted the network-access barrier for a large group of JavaScript developers whose work and ambition had begun to outgrow the tiny sandbox available to client developers. It should not be forgotten that the period of time extending from the introduction of JavaScript in 1995 to the present is nearly 20 years. Nearly a generation of developers has struggled trying to implement new ideas for network applications within an event-driven development environment known for, even defined by, its limitations. Overnight, Node removed those limitations. *If you clear paths, they will come*.
- Node provides an easy way to build scalable network programs, where network I/O is no longer a bottleneck. The real shift is not from another popular system to Node—it is away from the idea that expensive and complex resources are needed to build and maintain efficient applications demanding burstable concurrency. If a resilient and scalable network architecture can be achieved cheaply, freed resources can be directed to solving other pressing software challenges, such as parallelizing data filtering, scaling massively multiplayer games, building real-time trading platforms or collaborative document editors, even implementing live code changes in hot systems. Confidence breeds progress. *If you make it easy, they will come*.

Node arrived at a time when those building dynamic web pages had begun to run up against the limitations of servers not equipped to smoothly field many small, simultaneous requests. The software architect must now solve some interesting problems: what are the rules of "real time"—will the user be satisfied with "soon", or is "now" the only right response? And, what is the best way to design systems responsible for satisfying these user desires?

In this chapter we will investigate three standard techniques available to developers to use when constructing real-time network applications: AJAX, WebSockets, and **Server Sent Events** (**SSE**). Our goals for this chapter are to learn the benefits and drawbacks of each of these techniques, and to implement each technique with Node. Remembering that we are aiming to achieve a consistent architecture reflecting the evented-streams design of Node, we will also consider how well each technique lends itself to representation as a readable, writable, or duplex stream.

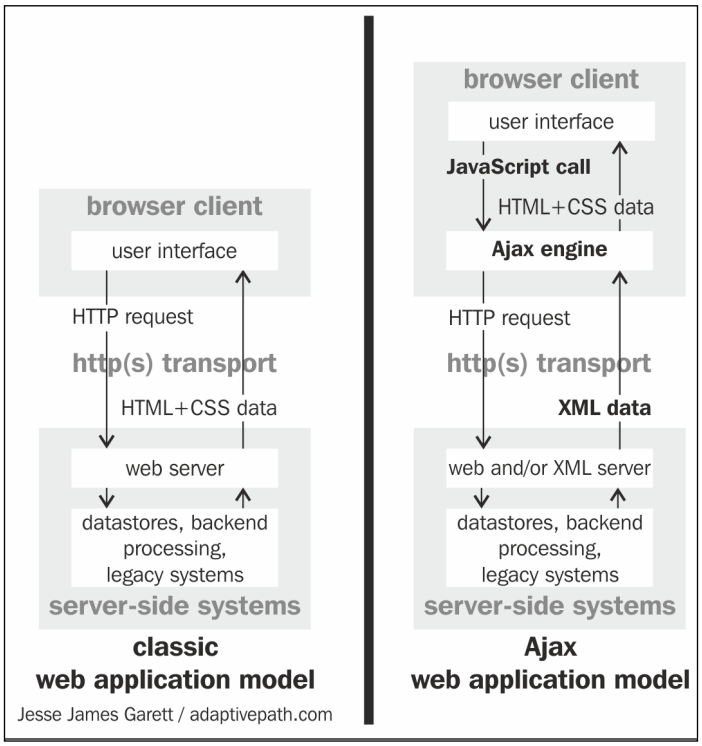
We will close this chapter with the construction of a collaborative code editor, which should demonstrate the opportunities Node provides for those seeking to build real-time groupware. As you work your way through the examples, and build your own applications, these are some questions that are worth asking yourself:

- What is the volume of messages I expect to transact per second? How many simultaneously connected clients are expected at peak times and at off-peak times?
- What is the average size and shape of the messages being transmitted? How big? How complex?
- Can I accept occasional communication breakdowns or dropped messages if this concession buys me lower average latency?
- Do I really need bidirectional communication, or is one side responsible for nearly all message volume? Do I need a complicated communication interface at all?
- What sorts of networks will my application run within? Will there be proxy servers between a client and my Node server? Which protocols are supported?
- Do I need a complex solution or will simple and straightforward, even slightly slower, solutions bring other benefits in the long run?

Introducing AJAX

In 2005, Jesse James Garrett published an article in which he tried to condense the changes he had been seeing in the way that websites were being designed into a pattern. After studying this trend, Garrett proposed that dynamically updating pages represented a new wave of software, resembling desktop software, and he coined the acronym "AJAX" to describe the technological concept powering such rapid movement toward "web applications".

This was the diagram he used to demonstrate the general pattern:



The original article can be found at `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications`.

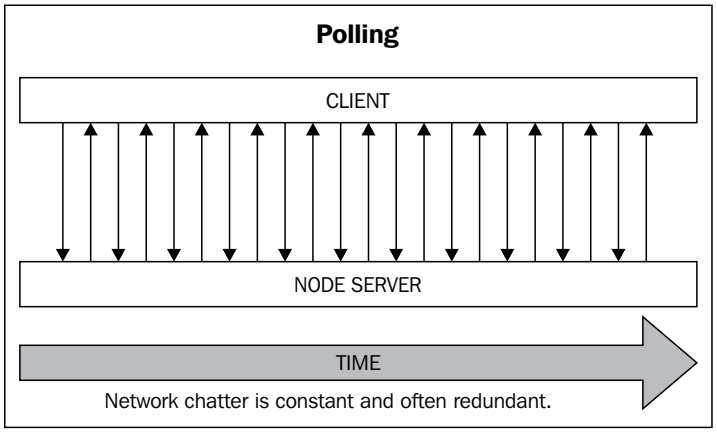
The "Ajax engine" Garrett's diagram referred to, had in fact existed in most common browsers by the year 2000, and even earlier in some. JavaScript implementations of the **XMLHttpRequest** (**XHR**) object in these browsers gave web pages the ability to request "chunks" of HTML or other data from servers. Partial updates could be dynamically applied to a web page, creating the opportunity for new kinds of user interfaces. For example, the latest pictures from an event could magically appear to a user, without that user actively requesting a page refresh, or clicking a "Next Picture" button.

More importantly, Garrett also understood how the synchronous, stateless world of the "old" Internet was becoming an asynchronous, stateful one. The conversation between clients and servers was no longer being derailed by sudden amnesia and could continue usefully for longer periods of time, sharing increasingly useful information. Garret saw this as a shift to a new generation of network software.

Responding to calls

If changes can be introduced into a web application without requiring a complete reconstruction of state and state display, updating client information becomes cheaper. The client and server can talk more often, regularly exchanging information. Servers can recognize, remember, and respond immediately to client desires, aided by reactive interfaces gathering user actions and reflecting the impact of those actions within a UI in near real time.

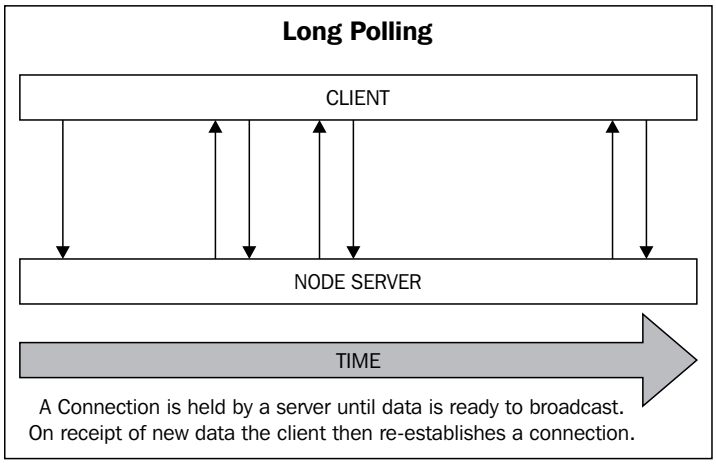
With AJAX, the construction of a multiuser environment supporting real-time updates to each client's view on the overall application state involves regular polling of server by clients checking for important updates:



The significant drawback to this method of polling for state is that many of these requests will be fruitless. The client has become a broken record, constantly asking for status updates regardless of whether those updates are available or forthcoming. When an application spends time or energy performing unnecessary tasks, there should exist some clear benefit to the user or the provider (or both) offsetting this cost. Additionally, each futile call adds to the cost of building up then tearing down HTTP connections.

Such a system can only take snapshots of the state at periodic intervals, and as that polling interval may increase to several seconds in an effort to reduce redundant network chatter, our awareness of state changes can begin to appear dull, just a little behind the latest news.

We saw a better solution in the previous chapter—long polling, the technique of letting a server hold on to a client connection until new data is available:

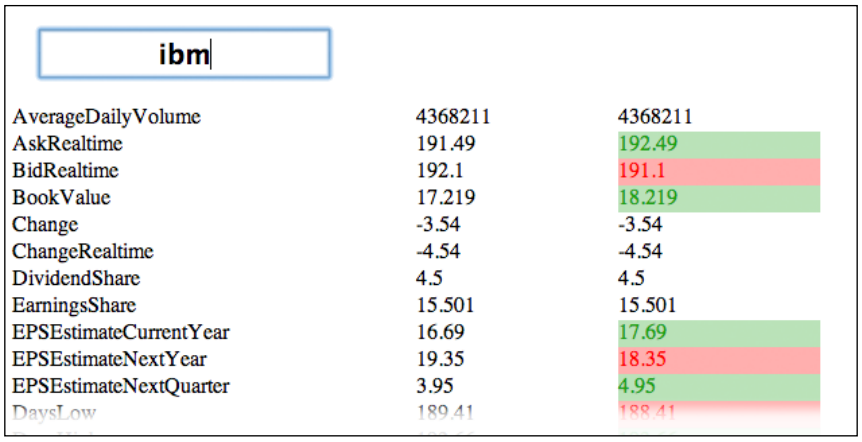


This improved AJAX technique does not fully escape the cost of building up and tearing down network connections, yet a significant reduction in the number of such costly operations is achieved. In general, AJAX fails to deliver a smooth, stream-like evented interface, requiring a great deal of attending services to persist state as connections are regularly broken and then re-established.

Nevertheless, AJAX remains a real option for some applications, in particular simple applications where the ideal polling interval is fairly well known, each poll standing a good chance of gathering useful results. Let's use Node to build a server able to communicate with a stock reporting service, and build a polling client that periodically requests this server to check for changes and report them.

Creating a stock ticker

Ultimately, we will create an application that allows clients to pick a stock and watch for changes in the data points related to that stock, such as its price, and to highlight positive or negative changes:



To create the client, our work is minimal. We need to simply poll our server every few seconds or so, updating our interface to reflect any data changes. Let's use jQuery as our AJAX library provider. To fetch JSON from a server using jQuery, you will normally do something like this:

```
var fetch = function() {
  $.getJSON("/service", function(data) {
    //  Do something with data
    updateDisplay(data);
    //  Call again in 5 seconds
    setTimeout(fetch, 5000);
  });
}
   fetch();
```

A Node server will receive this request for an update, perform some I/O (check a database, call an external service), and respond with `data`, which the client can use.

In our example, Node will be used to connect to the **Yahoo! Query Language** (**YQL**) service, which Yahoo! describes in this way:

> *The Yahoo! Query Language is an expressive SQL-like language that lets you query, filter, and join data across Web services. With YQL, apps run faster with fewer lines of code and a smaller network footprint.*

For example, if I wanted to read the latest stock quotes from Yahoo! Finance, my query might look like:

```
http://query.yahooapis.com/v1/public/yql?q=select * from yahoo.
finance.quotes where symbol in ("MSFT")
```

We will construct a Node server that listens for clients requesting an update to the data for a given stock symbol, such as "IBM". The Node server will then create a YQL query for that stock symbol and execute that query via `http.get`, packaging the received data nicely for the calling client and sending it back.

This package will also be assigned a new `callIn` property, indicating the number of milliseconds the client should wait before calling again. This is a useful technique to remember, as our stock data server will have a much better idea of the traffic conditions and the update frequency than the client will. Instead of a client blindly checking on a fixed schedule, our server can recalibrate this frequency after each call, even demanding that the client stop calling!

As this design, particularly the visual design, can be done through any number of ways, we will simply look at the core functionality necessary for our client, contained within the following `fetch` method:

```
var fetch = function() {
  var symbol = $("#symbol").val();
  $.getJSON("/?symbol=" + symbol, function(data) {
    if(!data.callIn) {
      return;
    }
    setTimeout(fetch, data.callIn);
    if(data.error) {
      return console.error(data.error);
    }
    var quote = data.quote;
    var keys = fetchNumericFields(quote);
…
    updateDisplay(symbol, quote, keys);
  });
};
```

Users on this page enter stock symbols into an input box with ID `#symbol`. This data is then fetched from our data service. In the preceding code, we see the service call being made via the `$.getJSON` jQuery method, the JSON data being received, and a `setTimeout` property being set using the `callIn` interval sent back by Node.

Our server is responsible for brokering the above client call with the Yahoo! Finance data service. Assuming that we have a properly configured the server that successfully receives stock symbols from clients, we need to open an HTTP connection to the YQL service, read any response, and return this data:

```
http.get(query, function(res) {
  var data = "";
  res
  .on('readable', function() {
    var d;
    while(d = this.read()) {
      data += d;
    }
  })
  .on('end', function() {
    var out = {};
    try {
      data = JSON.parse(data);
      out.quote = data.query.results.quote;
      out.callIn = 5000;
    } catch(e) {
      out = {
        error: "Received empty data set",
        callIn: 10000
      };
    }
    response.writeHead(200, {
      "Content-type" : "application/json"
    });
    response.end(JSON.stringify(out));
  });
}).on('error', function(e) {
  response.writeHead(200, {
    "Content-type" : "application/json"
  });
  response.end(JSON.stringify({
    error: "System Error",
    callIn: null
  }));
});
```

Here, we see a good example of why it is a good idea to let the server, the primary observer of state, modulate the frequency with which clients poll. If a successful data object is received, we set the poll interval (`callIn`) to about five seconds. Should an error occur, we increase that delay to 10 seconds. It is easy to see how we might do more, perhaps, throttling connections further if repeated errors occur. Given that, there will often be limits on the rate at which an application may make requests to an external service (such as limiting the number of calls that can be made in one hour), this is also a useful technique for ensuring that constant client polling doesn't exceed these rate limits.

AJAX is the original technique for creating real-time applications. It remains useful in some cases, but has been superseded by more efficient transports. As we leave this section, let's keep in mind some of the advantages and disadvantages of polling:

Pros	Cons
The theory and practice of REST is available, allowing more standardized communication	Making and breaking connections imposes a cost on network latency, especially if done very often
No need for any special protocol server, with polling easily implemented using a standard HTTP server	Clients must request data; servers are unable to unilaterally update clients as new data arrives
HTTP is well-known and consistently implemented	Even long-polling *doubles* the network traffic needed to maintain a persistent connection
	Data is blindly pushed and pulled, rather than smoothly broadcasted and listened for on channels

Let's move on now into a discussion of some newer protocols, in part designed to solve some of the issues we've found with AJAX: WebSockets and SSE.

Bidirectional communication with Socket.IO

We're already familiar with what sockets are. In particular, we know how to establish and manage TCP socket connections using Node, as well as how to pipe data through them bidirectionally or unidirectionally.

The W3C has proposed a socket API that allows browsers to communicate with a socket server over a persistent connection. Socket.IO is a library that facilitates the establishment of persistent socket connections for those developing with Node, providing both a Node-based socket server and an emulation layer for browsers that do not support the WebSocket API natively.

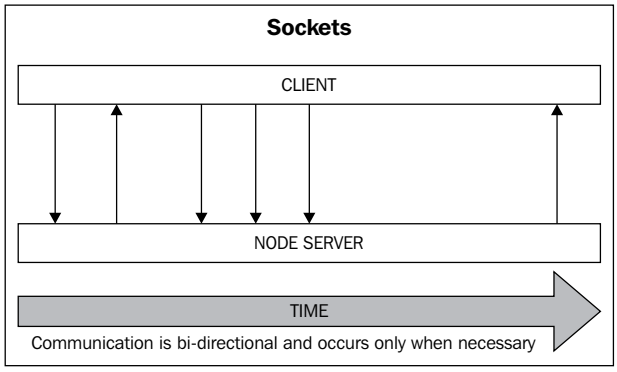
Let's first take a brief look at how the native WebSocket API is implemented, and how to build a socket server supporting this protocol using Node. We will then build a collaborative drawing application using socket.io with Node.

The full specification for the WebSocket API can be found at `http://www.w3.org/TR/websockets/`.

Documentation and installation instructions for Socket.IO can be found at `http://socket.io/`.

Using the WebSocket API

Socket communication is efficient, only occurring when one of the parties has something useful to say:



This lightweight model is an excellent choice for applications that require -frequency message passing between a client and a server, such as found in multiplayer network games or chat rooms.

According to the W3C, the WebSocket API is intended "to enable web applications to maintain bidirectional communications with server-side processes". Assuming that we have established a socket server running at `localhost:8080`, we can connect to this server from a browser containing the following line of JavaScript:

```
var conn = new WebSocket("ws://localhost:8080",['json', 'xml']);
```

WebSocket expects two arguments: a URL prefixed by the URI scheme `ws://`, and an optional subprotocol list, which can be an array or a single string of protocols that a server may implement.

To establish a secure socket connection, use the `wss://` prefix. As with HTTPS servers, an SSL certificate will need to be used when starting the server.

Once a socket request is made, the connection events `open`, `close`, `error`, and `message` can be handled by a browser:

```
<head>
  <title></title>
  <script>

  var conn = new WebSocket("ws://localhost:8080",'json');
  conn.onopen = function() {
    conn.send('Hello from the client!');
  };
  conn.onerror = function(error) {
    console.log('Error! ' + error);
  };
  conn.onclose = function() {
    console.log("Server has closed the connection!");
  };
  conn.onmessage = function(msg) {
    console.log('Received: ' + msg.data);
  };
  </script>
</head>
```

To implement a WebSocket server in Node, we will use the **ws** module created by Einar Otto Stangvik, which can be downloaded from `https://github.com/einaros/ws`. After installing ws using npm (`npm install ws`), establishing a Node socket server is straightforward:

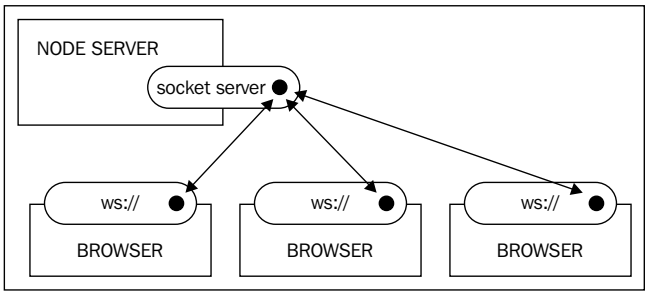
```
var SocketServer = require('ws').Server;
var wss = new SocketServer({port: 8080});
```

```
wss.on('connection', function(ws) {
    ws.on('message', function(message) {
        console.log('received: %s', message);
    });
    ws.send("You've connected!");
});
```

Here, we see how the server simply listens for the `connection` and `message` events from clients, responding as necessary. Should there be a need to terminate a connection (perhaps, if the client loses authorization), the server can simply emit a `close` event, which a client can listen for:

```
ws.close();
```

The general schematic for an application using the WebSocket API to create bidirectional communication therefore looks like this:



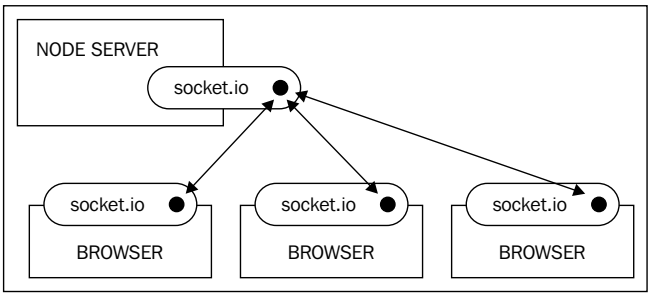
The native WebSocket browser implementation is here used to communicate with our custom Node socket server, which fields requests from the client as well as broadcasting new data or information to the client when necessary.

Socket.IO

As mentioned previously, Socket.IO aims to provide an emulation layer that will use the native WebSocket implementation in browsers that support it, reverting to other methods (such as long-polling) to simulate the native API in browsers that don't. This is an important fact to keep in mind: the real benefits of WebSockets will not exist for all client browsers.

Nevertheless, Socket.IO does a very good job of hiding browser differences and remains a good choice when the control flow made available by sockets is a desirable model of communication for your application.

In the WebSocket implementation previously given, it is clear that the socket server is independent of any specific client file. We wrote some JavaScript to establish a WebSocket connection on a client, independently running a socket server using Node. Unlike this native implementation, Socket.IO requires a custom client library to be installed on a server in addition to the `socket.io` server module:



Socket.IO can be installed using the npm package manager:

```
npm install socket.io
```

Setting up a client/server socket pairing is straightforward.

On the server:

```
var io = require('socket.io').listen(8080);
io.sockets.on('connection', function (socket) {
  socket.emit('broadcast', { message: 'Hi!' });
  socket.on('clientmessage', function (data) {
    console.log("Client said" + data);
  });
});
```

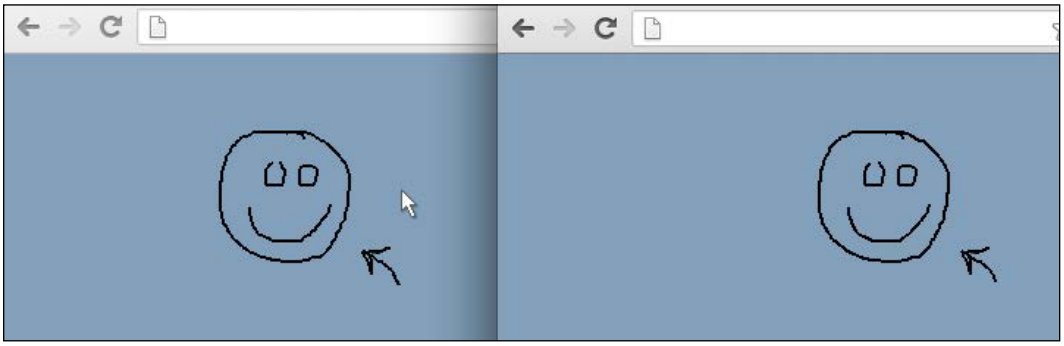
On the client:

```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io.connect('http://localhost');
  socket.on('broadcast', function(data) {
    console.log("Server sent: " + data);
    socket.emit('clientmessage', { message: 'ohai!' });
  });
</script>
```

We can see how both the client and the server are using the same file, `socket.io.js`. A server using Socket.IO handles the serving of the `socket.io.js` file to clients automatically when requested. It should also jump out how closely the Socket.IO API resembles a standard Node `EventEmitter` interface.

Drawing collaboratively

Let's create a collaborative drawing application using Socket.IO and Node. We want to create a blank canvas that will simultaneously display all the "pen work" being done by connected clients:



From the server end, there is very little to do. When a client updates coordinates by moving their mouse, the server simply broadcasts this change to all connected clients:

```
io.sockets.on('connection', function(socket) {
  var id = socket.id;
  socket.on('mousemove', function(data) {
    data.id = id;
    socket.broadcast.emit('moving', data);
  });
  socket.on('disconnect', function() {
    socket.broadcast.emit('clientdisconnect', id);
  });
});
```

Socket.IO automatically generates a unique ID for each socket connection. We will pass this ID along whenever new draw events occur, allowing the receiving clients to track how many users are connected. Similarly, when a client disconnects, all other clients are instructed to remove their references to this client. Later, we will see how this ID is used within the application UI to maintain a pointer representing all connected clients.

This is an excellent example of just how simple it is to create multiuser, Move onto one line network applications using Node and the packages created by the Node community. Let's break down what this server is doing.

Because we will need to deliver the HTML file that clients will use to draw, half of the server setup involves creating a static file server. For convenience, we'll use the node-static package, which can be downloaded from `https://github.com/cloudhead/node-static`. Our implementation will serve an `index.html` file to any client who connects.

Our Socket.IO implementation expects to receive `mousemove` events from clients, and its only task is to send to all connected clients these new coordinates, which it does by emitting a `moving` event through its `broadcast` method. As one client changes the canvas state by drawing a line, all clients will receive the information necessary to update their view of the canvas state in real time.

With the communication layer built, we now must create client views. As mentioned, each client will load an `index.html` file containing the necessary canvas element, and the JavaScript necessary to listen for `moving` events, as well the `socket.io` emitter broadcasting client draw events to our server:

```
<style type="text/css">
/* CSS styling for the pointers and canvas */
</style>
<script src="/socket.io/socket.io.js"></script>
<script src="/script.js"></script>
</head>
<body>
  <div id="pointers"></div>
  <canvas id="canvas" width="2000" height="1000"></canvas>
</body>
```

A `pointers` element is created to hold visible representations for the cursors of all connected clients, which will update as connected clients move their pointers and/or draw something.

Within the `script.js` file, we first set up event listeners on the `canvas` element, watching for the combination of `mousedown` and `mousemove` events indicating a draw action. Note how we create a time buffer of 50 milliseconds, delaying the broadcast of each draw event, slightly reducing the resolution of drawings but avoiding an excessive number of network events:

```
var socket = io.connect("/");
var prev = {};
var canvas  = document.getElementById('canvas');
var context = canvas.getContext('2d');
```

```
var pointerContainer = document.getElementById("pointers");
var pointer = document.createElement("div");
pointer.setAttribute("class", "pointer");
var drawing  = false;
var clients  = {};
var pointers = {};
var drawLine = function(fromx, fromy, tox, toy) {
  context.moveTo(fromx, fromy);
  context.lineTo(tox, toy);
  context.stroke();
};
canvas.onmouseup   =
canvas.onmousemove =
canvas.onmousedown = function(e) {
  switch(e.type) {
    case "mouseup":
      drawing = false;
    break;
    case "mousemove":
      if(now() - lastEmit > 50) {
        socket.emit('mousemove', {
          'x'       : e.pageX,
          'y'       : e.pageY,
          'drawing': drawing
        });
        lastEmit = now();
      }
      if(drawing) {
        drawLine(prev.x, prev.y, e.pageX, e.pageY);

        prev.x  = e.pageX;
        prev.y  = e.pageY;
      }
    break;
    case "mousedown":
      drawing = true;
      prev.x  = e.pageX;
      prev.y  = e.pageY;
    break;
    default:
    break;
  }
}
```

Whenever a draw action occurs (a combination of a `mousedown` and a `mousemove` event), we draw the requested line on the client's machine, and then broadcast these new coordinates to our Socket.IO server via `socket.emit('mousemove', …)`, remembering to pass along the `id` value of the drawing client. The server in turn will broadcast them via `socket.broadcast.emit('moving', data)`, allowing client listeners to draw equivalent lines on their `canvas` element:

```
socket.on('moving', function(data) {
  if(!clients.hasOwnProperty(data.id)) {
    pointers[data.id] = pointerContainer.appendChild(pointer.
cloneNode());
  }
  pointers[data.id].style.left = data.x + "px";
  pointers[data.id].style.top = data.y + "px";

  if(data.drawing && clients[data.id]){
    drawLine(clients[data.id].x, clients[data.id].y, data.x, data.y);
  }
  clients[data.id] = data;
  clients[data.id].updated = now();
});
```

Within this listener, a client will establish a new client pointer if the sent client ID has not been see previously, and animate both the drawing of a line and the client pointer, creating the effect of multiple cursors drawing distinct lines within a single client view.

Recalling the `clientdisconnect` event we track on our server, we also enable clients to listen for these disconnects, removing references to lost clients from both the view (visual pointer) and our `clients` object:

```
socket.on("clientdisconnect", function(id) {
  delete clients[id];
  if(pointers[id]) {
    pointers[id].parentNode.removeChild(pointers[id]);
  }
});
```

Socket.IO is an excellent tool to consider when building interactive, multiuser environments where continuous rapid bidirectional data transfer is necessary.

Now, take a look at the pros and cons of Socket.IO:

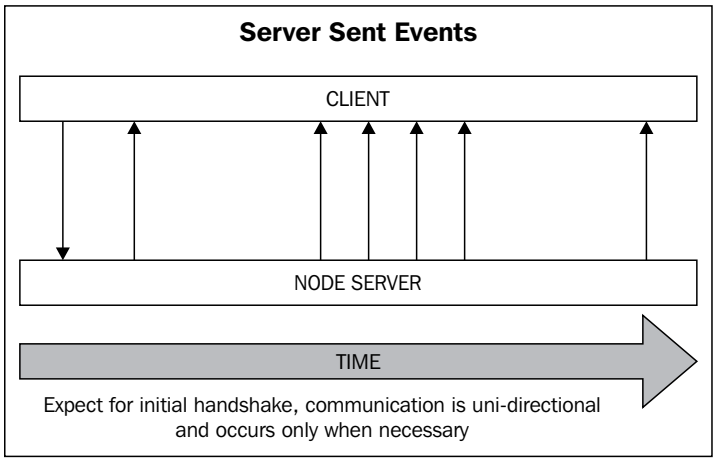
Pros	Cons
Rapid bidirectional communication essential to real-time games, collaborative editing tools, and other applications	The number of allowed persistent socket connections can be limited on the server side or anywhere in between
Lower overhead than standard HTTP protocol request, lowering the price of sending a package across the network	Many browsers do not natively support web sockets, forcing emulation libraries, such as Socket.IO, to resort to long-polling or other inefficient techniques
The evented, streaming nature of sockets fits conceptually with the Node architecture—clients and servers are simply piping data back and forth through consistent interfaces	Requires a custom protocol server, and often a custom client library
	Many proxies and reverse-proxies are known to confound socket implementations, leading to lost clients

Another excellent socket library for Node development is **SockJS** (`https://github.com/sockjs`).

Listening for Server Sent Events

SSE are uncomplicated and specific. They are to be used when the majority of data transfer proceeds unidirectionally from a server to clients. A traditional and similar concept is the "push" technology. SSE passes text messages with simple formatting. Many types of applications passively receive brief status updates or data state changes. SSE is an excellent fit for these types of applications.

Like WebSocket, SSE also eliminates the redundant chatter of AJAX. Unlike WebSocket, an SSE connection is only concerned with broadcasting data from servers to connected clients:



A client connects to a server supporting SSE by passing the `EventSource` constructor a path:

```
var eventSource   = new EventSource('/login');
```

This instance of `EventSource` will now emit subscribable data events whenever new data is received from the server.

Using the EventSource API

The way in which `EventSource` instances emit subscribable data events whenever new data is received from the server is like the way `Readable` streams emit data events in Node, as we can see in this example client:

```
<script>
  var eventSource = new EventSource('/login');
  eventSource.addEventListener('message', function(broadcast) {
    console.log("got message: " + broadcast);
  });
  eventSource.addEventListener('open', function() {
    console.log("connection opened");
  });
  eventSource.addEventListener('error', function() {
    console.log("connection error/closed");
  });
</script>
```

An `EventSource` instance emits three default events:

- `open`: When a connection is successfully opened, this event will fire
- `message`: The handler assigned to this event will receive an object whose `data` property contains the broadcast message
- `error`: This fires whenever a server error occurs, or the server disconnects or otherwise severs its connection with this client

Forming part of the standard HTTP protocol, a server responsive to SSE requests requires minimal configuration. The following server will accept `EventSource` bindings and broadcast the current date to the bound client every second:

```
var http = require("http");
var url = require("url");
http.createServer(function(request, response) {
  var parsedURL = url.parse(request.url, true);
  var pathname  = parsedURL.pathname;
  var args      = pathname.split("/");
  var method    = args[1];
    if(method === "login") {
    response.writeHead(200, {
      "Content-Type": "text/event-stream",
      "Cache-Control": "no-cache",
      "Connection": "keep-alive"
    });
    response.write(":" + Array(2049).join(" ") + "\n");
    response.write("retry: 2000\n");

    response.on("close", function() {
      console.log("client disconnected");
    });
    setInterval(function() {
      response.write("data: " + new Date() + "\n\n");
    }, 1000);
    return;
  }
}).listen(8080);
```

This server listens for requests and selects those made on the path `/login`, which it interprets as a request for an `EventSource` binding. Establishing an `EventSource` connection is simply a matter of responding to the request with a `Content-Type` header of `text/event-stream`. Additionally we indicate that the client's `Cache-Control` behavior should be set to `no-cache`, as we expect a lot of original material on this channel.

From the point of connection the `response` object of this client will remain an open pipe that messages can be sent through using `write`. Let's look at the next two lines:

```
response.write(":" + Array(2049).join(" ") + "\n");
response.write("retry: 2000\n");
```

This first write is adjusting for an XHR implementation feature in some browsers, which ultimately requires all SSE streams to be prefixed by a 2-KB padding. This `write` action need happen only once, and has no relevance to subsequent messages.

To read more about why this 2-KB padding is necessary, visit `http://blogs.msdn.com/b/ieinternals/archive/2010/04/06/comet-streaming-in-internet-explorer-with-xmlhttprequest-and-xdomainrequest.aspx?PageIndex=1`.

One of the advantages of SSE is that clients will automatically try to reconnect with the server, should that connection be severed. The number of milliseconds before retrying will vary from client to client, and can be controlled using the `retry` field, which we use here to set a two-millisecond retry interval.

Finally, we listen for the client's `close` event, which fires when a client disconnects, and begins broadcasting the time on a one-second interval:

```
setInterval(function() {
  response.write("data: " + new Date() + "\n\n");
}, 1000);
```

A website might bind to this time server and display the current server time:

```
<html>
<head>
  <script>
  var ev = new EventSource('/login');
  ev.addEventListener("message", function(broadcast) {
    document.getElementById("clock").innerHTML = broadcast.data;
  });
  </script>
</head>
<body>
  <div id="clock"></div>
</body>
</html>
```

Because the connection is one-way, any number of services can be set up as publishers very easily, with clients binding individually to these services via new `EventSource` instances. Server monitoring, for example, could be achieved easily by modifying the above server such that it periodically sends the value of `process.memoryUsage()`. As an exercise, use SSE to re-implement the stocks service we covered earlier in the section on AJAX.

The EventSource stream protocol

Once a server has established a client connection, it may now send new messages across this persistent connection at any time. These messages consist of one or more lines of text, demarcated by one or several of the following four fields:

- `event`: This is an event type. Messages sent without this field will trigger the client's general `EventSource` event handler for any message. If set to a string such as "latestscore", the client's `message` handler will *not* be called, with handling being delegated to a handler bound using `EventSource.addEventListener('latestscore'…)`.
- `data`: This is the message being sent. This is always of the `String` type, though it can usefully transport objects passed through `JSON.stringify()`.
- `id`: If set, this value will appear as the `lastEventID` property of the sent message object. This can be useful for ordering, sorting, and other operations on the client.
- `retry`: The reconnection interval, in milliseconds.

Sending messages involves composing strings containing relevant field names and ending with newlines. These are all valid messages:

```
response.write("id:" + (++message_counter) + "\n");
response.write("data: I'm a message\n\n");
response.write("retry: 10000\n\n");
response.write("id:" + (++message_counter) + "\n");
response.write("event: stock\n");
response.write("data: " + JSON.stringify({price: 100, change: -2}) +
"\n\n");
response.write("event: stock\n");
response.write("data: " + stock.price + "\n");
response.write("data: " + stock.change + "\n");
response.write("data: " + stock.symbol + "\n\n");
response.write("data: Hello World\n\n");
```

We can see that multiple `data` fields can be set as well. An important thing to note is the double-newline (`"\n\n"`) to be sent after the *final* data field. Previous fields should just use a single newline.

The default `EventSource` client events (`open`, `message`, and `close`) are sufficient for modeling most application interfaces. All broadcasts from the server are caught within the solitary `message` handler, which takes responsibility for routing the message or otherwise updating the client, in the same way that event delegation would work when working with events in the DOM using JavaScript.

This system may not be ideal in cases where many unique message identifiers are needed, overwhelming a single handling function. We can use the `event` field of SSE messages to create custom event names that can be individually bound by a client, neatly separating concerns.

For example, if two special events `actionA` and `actionB` are being broadcast, our server would structure them like this:

```
event: actionA\n
data: Message A here\n\n

event: actionB\n
data: Message B here\n\n
```

Our client would bind to them in the normal way, as shown in the following code snippet:

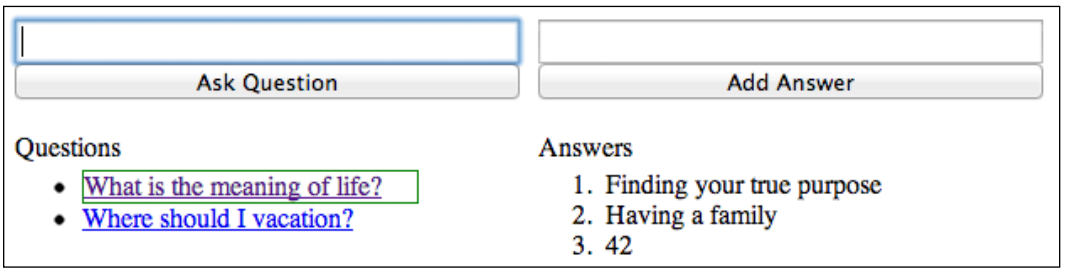
```
ev.addEventListener("actionA", function(broadcast) {
  console.log(broadcast.data);
});

ev.addEventListener("actionB", function(broadcast) {
  console.log(broadcast.data);
});
```

In cases where a single message handling function is becoming too long or too complex, consider uniquely named messages and handlers.

Asking questions and getting answers

What if we wanted to create an interface to interests? Let's build an application enabling any number of people to ask and/or answer questions. Our users will join the community server, see a list of open questions and answers to those questions, and get real-time updates whenever a new question or answer is added. There are two key activities to model:

- Each client must be notified whenever another client asks a question or posts an answer
- A client can ask questions or supply answers



Where would the greatest amount of change happen in a large group of simultaneous contributors to this community?

Any individual client can potentially ask a few questions or provide a few answers. Clients will also select questions and have the answers displayed to them. We will need to satisfy merely a handful of client-to-server requests, such as when sending a new question or answer to the server. Most of the work will be in satisfying client requests with data (a list of answers to a question) and broadcasting application state changes to *all* connected clients (new question added; new answer given). The one-to-many relationship existing for clients within such collaborative applications implies that a single client broadcast may create a number of server broadcasts equal to the number of connected clients—1 to 10 K, or more. SSE is a great fit here, so let's get started.

The three main operations for this application are as follows:

- Asking a question
- Answering a question
- Selecting a question

Either of these actions will change the application state. As this state must be reflected across all clients, we will store the state of our application on our server—all questions, answers, and the relationships of clients to these data objects. We will also need to uniquely identify each client. Normally, one would use a database to persist some of this information, but for our purposes we will simply store this data in our Node server:

```
var USER_ID     = 1e10;
var clients     = {};
var clientQMap  = {};
var questions   = {};
var answers     = {};
var removeClient = function(id) {
  if(id) {
    delete clients[id];
    delete clientQMap[id];
  }
}
```

In addition to the `questions` and `answers` storage objects, we will also need to store client objects themselves—clients are assigned a unique ID that can be used to look up information (such as the client's socket) when broadcasts are made.

We only want to broadcast answer data to clients that have expressed an interest in the specific question—as client UIs are only displaying answers for a single question, we of course would not broadcast answers to clients indiscriminately. For this reason, we keep a `clientQMap` object, which maps a question to all clients listening to that question, by the ID.

The `removeClient` method is straightforward: when a client disconnects, the method removes its data from the pool. We'll see this again later.

With this setup in place, we next need to build our server to respond the `/login` path, which is used by `EventSource` to grab a connection. This service is responsible for configuring a proper event-stream for clients, storing this `Response` object for later use, and assigning the user a unique identifier, which will be used on future client requests to identify the client and fetch that client's communication socket:

```
http.createServer(function(request, response) {
  var parsedURL   = url.parse(request.url, true);
  var pathname    = parsedURL.pathname;
  var args        = pathname.split("/");
  //  Lose initial null value
  args.shift();
  var method    = args.shift();
```

```
    var parameter = decodeURIComponent(args[0]);
    var sseUserId = request.headers['_sse_user_id_'];
      if(method === "login") {
      response.writeHead(200, {
        "Content-Type": "text/event-stream",
        "Cache-Control": "no-cache"
      });
      response.write(":" + Array(2049).join(" ") + "\n"); // 2kB
      response.write("retry: 2000\n");
      removeClient(sseUserId);
      //  A very simple id system. You'll need something more secure.
      sseUserId = (USER_ID++).toString(36);
      clients[sseUserId] = response;
      broadcast(sseUserId, {
        type    : "login",
        userId  : sseUserId
      });
      broadcast(sseUserId, {
        type      : "questions",
        questions : questions
      });
      response.on("close", function() {
        removeClient(sseUserId);
      });

      // In order to keep the connection alive we send a "heartbeat"
  every 10 seconds.
      https://bugzilla.mozilla.org/show_bug.cgi?id=444328
      setInterval(function() {
        broadcast(sseUserId, new Date().getTime(), "ping");
      }, 10000);
      return;
    }
```

After establishing request parameters, our servers check the request for a `_sse_user_id_` header, which is the unique string that is assigned to a user within `/login` on the initial `EventSource` bind:

```
sseUserId = (USER_ID++).toString(36);
clients[sseUserId] = response;
```

This ID is then sent to the client via an immediate broadcast, an opportunity we use to send along the current batch of questions:

```
broadcast(sseUserId, sseUserId, "login");
```

The client is now responsible for passing along this ID whenever it makes a call. By listening for the `/login` event and storing the ID that is passed, a client can self-identify when making HTTP calls:

```
evSource.addEventListener('login', function(broadcast) {
  USER_ID = JSON.parse(broadcast.data);
});
var xhr = new XMLHttpRequest();
xhr.open("POST", "/...");
xhr.setRequestHeader('_sse_user_id_', USER_ID);
...
```

Recall that we have just created a unidirectional event-stream from our server to our client. This channel is used to communicate with clients—not `response.end()` or similar. The `broadcast` method, referenced in `/login`, accomplishes this task of broadcasting stream events as shown in the following code:

```
var broadcast = function(toId, msg, eventName) {
    if(toId === "*") {
      for(var p in clients) {
        broadcast(p, msg);
      }
      return;
    }
    var clientSocket = clients[toId];
    if(!clientSocket) {
      return;
    }
    eventName && clientSocket.write("event: " + eventName + "\n");
    clientSocket.write("id: " + (++UNIQUE_ID) + "\n");
    clientSocket.write("data: " + JSON.stringify(msg) + "\n\n");
  }
```

Scan this code from the bottom up. Note how the primary purpose of `broadcast` is to take a client ID, look up that client's event-stream, and write to it, accepting a custom event name if needed. However, as we will regularly broadcast to *all* connected clients, we allow for a special `*` flag to indicate mass broadcast.

Everything is now set up, requiring only the definition of services for the three main operations for this application: adding new questions and answers, and remembering the question each client is following.

When questions are asked, we ensure that the question is unique, add it to our `question` collection, and tell everyone the new question list:

```
if(method === "askquestion") {
  //  Already asked?
  if(questions[parameter]) {
    return response.end();
  }
  questions[parameter] = sseUserId;
  broadcast("*", {
    type  : "questions",
    questions  : questions
  });
  return response.end();
}
```

Handling answers is nearly identical, except that here we want to broadcast new answers only to clients asking the right questions:

```
if(method === "addanswer") {

...
  answers[curUserQuestion] = answers[curUserQuestion] || [];
  answers[curUserQuestion].push(parameter);
  for(var id in clientQMap) {
    if(clientQMap[id] === curUserQuestion) {
      broadcast(id, {
        type     : "answers",
        question : curUserQuestion,
        answers  : answers[curUserQuestion]
      });
    }
  }
  return response.end();
}
```

Finally, we store changes to the client's interests by updating `clientQMap`:

```
if(method === "selectquestion") {
  if(parameter && questions[parameter]) {
    clientQMap[sseUserId] = parameter;
    broadcast(sseUserId, {
```

```
        type      : "answers",
        question  : parameter,
        answers   : answers[parameter] ? answers[parameter] : []
      });
    }
    return response.end();
  }
```

While we won't go too deeply into the client-side HTML and JavaScript necessary to render this interface, we will look at how some of the core events would be handled.

Assuming a UI rendered in HTML, which lists answers on one side and questions on the other, containing forms for adding new questions and answers, as well as for selecting questions to follow, our client code is very lightweight and easy to follow. After negotiating the initial `/login` handshake with our server, this client need simply sends new data via HTTP when submitted. The handling of server responses is neatly encapsulated into three events, making for easy to follow event stream handling:

```
var USER_ID    = null;
var evSource   = new EventSource('/login');
var answerContainer = document.getElementById('answers');
var questionContainer = document.getElementById('questions');
var showAnswer = function(answers) {
  answerContainer.innerHTML = "";
  var x = 0;
  for(; x < answers.length; x++) {
    var li = document.createElement('li');
    li.appendChild(document.createTextNode(answers[x]));
    answerContainer.appendChild(li);
  }
}
var showQuestion = function(questions) {
  questionContainer.innerHTML = "";
  for(var q in questions) {
    //... show questions, similar to #showAnswer
  }
}
evSource.addEventListener('message', function(broadcast) {
  var data = JSON.parse(broadcast.data);
  switch(data.type) {
    case "questions":
      showQuestion(data.questions);
    break;
    case "answers":
      showAnswer(data.answers);
```

```
      break;
      case "notification":
        alert(data.message);
      break;
      default:
        throw "Received unknown message type";
      break;
    }
});
evSource.addEventListener('login', function(broadcast) {
  USER_ID = JSON.parse(broadcast.data);
});
```

This interface needs only to wait for new question and answer data, and display it in lists. Three callbacks are enough to keep this client up-to-date, regardless of how many different clients update the application's state.

Pros	Cons
Lightweight: By using the native HTTP protocol an SSE server can be created with a couple of simple headers	Inconsistent browser support requires a custom library for client-to-server communication, where unsupported browsers will normally long-poll
Able to send data to a client unilaterally, without requiring matching client calls	One way only: Does not bring its advantages to cases where bidirectional communication is needed
Automatic reconnection of dropped connections, making SSE a reliable network binding	Server must send a "heartbeat" every 10 seconds or so in order to keep the connection alive
Simple, easily customizable, and an easy-to-understand messaging format	

`EventSource` is not supported by all browsers (in particular, IE). An excellent emulation library for SSE can be found at `https://github.com/Yaffle/EventSource`.

Building a collaborative document editing application

Now that we've examined various techniques to consider when building a collaborative application, let's put together a collaborative code editor using one of the most exciting contributions to Node to arrive in the last year: **Operational transformation** (**OT**).

For our discussion here, OT will be understood as a technology that allows many people to edit the same document concurrently—collaborative document editing. Google described their (now defunct) Wave project in the following way:

> *Collaborative document editing means multiple editors are able to edit a shared document at the same time. It is live and concurrent when a user can see the changes another person is making, keystroke by keystroke. Google Wave offers live concurrent editing of rich text documents.*

Source: `http://www.waveprotocol.org/whitepapers/operational-transform`

One of the engineers involved in the Wave project was Joseph Gentle, and Mr. Gentle was kind enough to write a module bringing OT technology to the Node community, named **ShareJS** (`www.sharejs.org`). We are going to use this module to create an application that allows anyone to create a new collaboratively editable document.

This example follows (and liberally borrows from) the many examples contained in the ShareJS GitHub repository. To delve deeper into the possibilities of ShareJS, visit `https://github.com/share/ShareJS`.

To begin with we will need a code editor to bind our OT layer to. For this project, we will use the excellent Ace editor, which can be cloned from `https://github.com/ajaxorg/ace`.

Establishing an Ace editor doesn't require much more than cloning the repository and writing the following HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Editor</title>
  <style type="text/css" media="screen">
    body {
```

```
            overflow: hidden;
        }
        #editor {
            margin: 0;
            position: absolute;
            top: 0;
            bottom: 0;
            left: 0;
            right: 0;
        }
    </style>
</head>
<body>
    <pre id="editor">function hello(items) {
      alert("Hello World!");
    }</pre>
    <script src="/ace/ace.js" charset="utf-8"></script>
    <script src="/ace/mode-javascript.js"></script>
    <script src="/ace/theme-eclipse.js"></script>
    <script>
      var editor = ace.edit("editor");
      editor.getSession().setMode(new (require("ace/mode/javascript").
Mode));
      editor.setTheme("ace/theme/eclipse");
    </script>
</body>
</html>
```

Let's transform this flat editor into a collaborative one. We first add the following to the `<script>` collection:

```
        <script src="/channel/bcsocket.js"></script>
        <script src="/share/share.js"></script>
        <script src="/share/ace.js"></script>
```

ShareJS uses the `bcsocket` library, which is an implementation of a Google Browser channel server for Node. This library acts as a socket library through which clients and ShareJS servers can communicate. We then load the ShareJS library itself, and finally the binding file that enables Ace documents to notify ShareJS of transformations, and to be transformed dynamically.

Once we have these libraries at the ready, we can bind to a ShareJS server:

```
sharejs.open("documentname", 'text', function(error, doc) {
  if(error) {
    throw(error);
```

```
    }
    if(doc.created) {
      doc.insert(0, "var helloWorld = function() { \n    alert('Hello
World!'); \n};");
    }
    doc.attach_ace(editor);
    editor.setReadOnly(false);
});
```

Here, we are making a request to a ShareJS server asking for the creation of a document of a certain name. If the creation of that OT document is successful, add some text to that document and bind our Ace editor to that OT document stream.

All that is left to do is create the Node server exposing the functionality of OT to the client. We will need some way of persisting documents through time, as these large data objects would likely overwhelm a Node process if all were stored in memory. In our implementation we will use Redis. Additionally, we will build this server using the Express framework, which can be easily configured to run applications using ShareJS.

As we've seen in our earlier examples, this entire server can be built using surprisingly little code:

```
// ShareJS uses CoffeeScript, requiring this compiler.
require('coffee-script');
var express = require('express');
var sharejs = require('share');
var server = express();
server.use(express.static(__dirname + '/'));
var options = {
  db: {type: 'none'},
  auth: function(client, action) {
      action.accept();
  }
};
// Let's try and enable redis persistence if redis is installed...
try {
  require('redis');
  options.db = {type: 'redis'};
} catch(e) {}
console.log("Server started");
// Bind sharejs interfaces to our server
sharejs.server.attach(server, options);
server.listen(8080);
process.title = 'editor demo'
```

The functionality of this server is straightforward. When documents are requested (via a client call to `sharejs.open`), this server will open up a persistent connection to the requesting client, and will share any changes other clients bound to the same document may make to it. As mentioned earlier, Redis is used to persist data.

The one thing to note is the `auth` function passed in the ShareJS option object. We here accept all incoming connections. If we were instead interested in being able to reject connections based on some authorization scheme, we would transform this function into something like the following code:

```
auth: function(client, action) {
if(client.sessionId in authenticatedSessions) {
    return action.accept();
  }
  action.reject();
}
```

The `client` parameter sent to this function contains a full breakdown of request headers and session data, which can be used to validate clients in ways we've used throughout this book. The `action` parameter will contain an object describing the type of OT action requested, such as `update` or `read`, as well as the name of the document requested. These further values can be used to be even more discerning with authentication—some clients may be allowed to read but not to make changes, for example.

This server can now be used to share document state across all clients requesting identically named documents, facilitating collaborative editing.

Another OT library to consider is `https://github.com/Operational-Transformation/ot.js/`.

Summary

In this chapter we've gone over three of the major strategies employed when building real-time applications: AJAX, WebSocket, and SSE. We've shown that non-trivial collaborative applications can be developed with very little code by using Node. We've also seen how some of these strategies enable the modeling of client/server communication as an evented data stream interface. We've considered the pros and cons of these various techniques, and we've gone through some clear examples of the best places to use each one.

Additionally, we've shown how client identifiers and state date can be built and managed within a Node server, such that state changes can be safely encapsulated in a central location and broadcasted out to many connected clients safely and predictably. Demonstrating the quality of the modules being developed with the Node community, we created a collaborative code editing system through the use of operational transformation.

In the next chapter we will be looking at how to coordinate the efforts of multiple Node processes running simultaneously. Through examples, we will learn how to achieve parallel processing with Node, from spawning many child processes running Unix programs to creating clusters of load-balancing Node socket servers.

7
Utilizing Multiple Processes

"*It is a very sad thing that nowadays there is so little useless information.*"

– Oscar Wilde

The importance of I/O efficiency is not lost on those witnessing the rapidly increasing volume of data being produced within a growing number of applications. User-generated content (blogs, videos, tweets, posts) is becoming the premier type of internet content, and this trend has moved in tandem with the rise of social software, where mapping the intersections between content generates an exponential rise in yet another level of data.

A number of data silos, such as Google, Facebook, and hundreds of others, expose their data to the public through an API, often for free. These networks each gather astounding volumes of content, opinions, relationships, and so forth from their users, data further augmented by market research and various types of traffic and usage analysis. Most of these APIs are two-way, *gathering* and *storing* data uploaded by their members as well as *serving* that data.

Node has arrived during this period of data expansion. In this chapter we will investigate how Node addresses this need for sorting, merging, searching, and otherwise manipulating large amounts of data. Fine-tuning your software, so that it can process large amounts of data safely and inexpensively, is critical when building fast and scalable network applications.

We will deal with specific scaling issues in the next chapter. In this chapter we will study some best practices when designing systems where multiple Node processes work together on large volumes of data.

As part of that discussion, we will be investigating strategies for parallelism when building data-heavy applications, focused on how to take advantage of multiple CPU environments, use multiple workers, and leverage the OS itself to achieve the efficiency of parallelism. The process of assembling applications out of these contained and efficient processing units will be demonstrated by example.

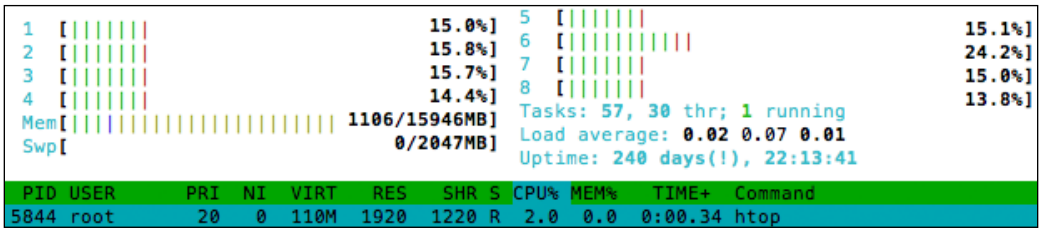
As noted in *Chapter 5*, *Managing Many Simultaneous Client Connections*, concurrency is not the same as parallelism. The goal of concurrency is good structure for programs, where modeling the complexities inherent in juggling multiple simultaneous processes is simplified. The goal of parallelism is to increase application performance by sharing parts of a task or computation across many workers. It is useful to recall Clinger's vision of "…dozens, hundreds or even thousands of independent microprocessors, each with its own local memory and communications processor, communicating via a high-performance communications network."

We've already discussed how Node helps us reason about non-deterministic control flow. Let's also recall how Node's designers follow the Rule Of Modularity, which encourages us to write simple parts connected by clean interfaces. This rule leads to a preference for simple networked processes communicating with each other using a common protocol. An associated rule is the **Rule of Simplicity**, stated as follows:

> *Developers should design for simplicity by looking for ways to break up program systems into small, straightforward cooperating pieces. This rule aims to discourage developers' affection for writing "intricate and beautiful complexities" that are in reality bug prone programs.*
>
> —`http://en.wikipedia.org/wiki/Unix_philosophy`

It is good to keep this rule in mind as we proceed through this chapter. To tame expanding data volume we can build enormous, complex, and powerful monoliths in the hope that they will remain big and powerful enough. Alternatively, we can build small and useful units of processing that can be combined into a single processing team of any size, not unlike the way that supercomputers can be built out of many thousands or millions of cheap commodity processors.

A process viewer will be useful while working through this chapter. A good one for Unix systems is **htop**, which can be downloaded from `http://htop.sourceforge.net/`. This tool provides, among other things, a view into CPU and memory usage; here we see how load is spread across all eight cores:



Node's single-threaded model

Taken in its entirety, the Node environment usefully demonstrates *both* the efficiency of multithreaded parallelism and an expressive syntax amenable to applications featuring high concurrency. Using Node does not constrain the developer, the developer's access to system resources, or the types of applications the developer might like to build.

Nevertheless, a surprising number of persistent criticisms of Node are based on this misunderstanding. As we'll see, the belief that Node is not multithreaded and is, therefore, slow, or not ready for prime time, simply misses the point. JavaScript is single-threaded; the Node stack is not. JavaScript represents the language used to coordinate the execution of several multithreaded C++ processes, even the bespoke C++ add-ons created by you, the developer. Node provides JavaScript, run through V8, primarily as a tool for modeling concurrency. That, additionally, one can write an entire application using just JavaScript is simply another benefit of the platform. You are never stuck with JavaScript—you may write the bulk of your application in C++ if that is your choice.

In this chapter we will attempt to dismantle these misunderstandings, clearing the way for optimistic development with Node. In particular, we will study techniques for spreading effort across cores, processes, and threads. For now, this section will attempt to clarify how much a single thread is capable of (hint: it's usually all you need).

The benefits of single-threaded programming

You will be hard-pressed to find any significant number of professional software engineers working on enterprise-grade software willing to deny that multithreaded software development is painful. However, *why* is it so hard to do well?

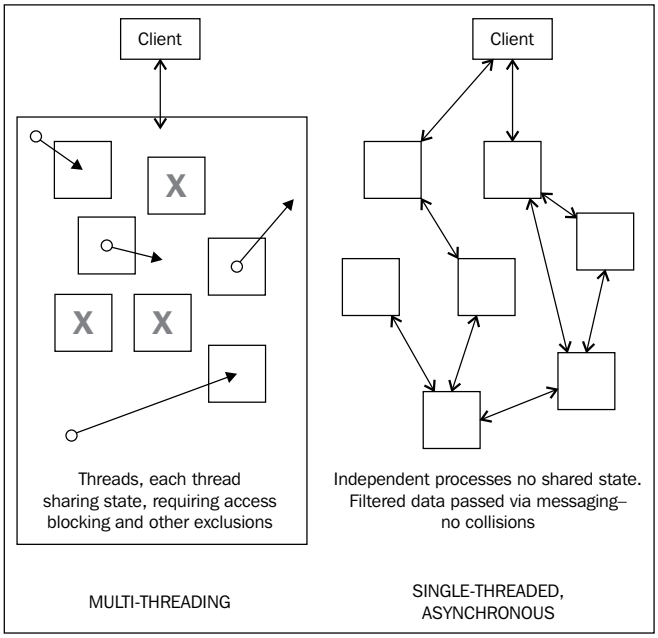
It is not that multithreaded programming is difficult per se—the difficultly lies in the complexity of thread synchronization. It is very difficult to build high concurrency using the thread model, especially models in which the state is shared. Anticipating every way that an action taken in one thread might affect all the others is nearly impossible once an application grows beyond the most basic of shapes. Entanglements and collisions multiply rapidly, sometimes corrupting shared memory, sometimes creating bugs nearly impossible to track down.

Node's designers chose to recognize the speed and parallelization advantages of threads without demanding that developers did the same. In particular, Node's designers wanted to save developers from managing the difficulties that accompany threaded systems:

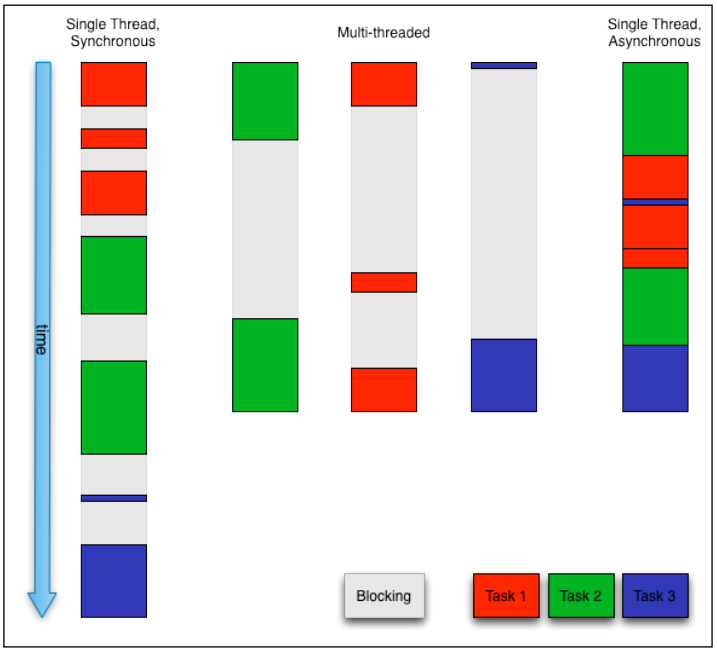
- Shared memory and the locking behavior leads to systems that are very difficult to reason about as they grow in complexity.
- Communication between tasks requires the implementation of a wide range of synchronization primitives, such as mutexes and semaphores, condition variables and so forth. An already challenging environment requires highly complex tools, expanding the level of expertise necessary to complete even relatively simple systems.
- Race conditions and deadlocks are common pitfalls in these sorts of systems. Contemporaneous read and write operations within a shared program space lead to problems of sequencing, where two threads may be in an unpredictable "race" for the right to influence a state, event, or other key system characteristic.
- Because maintaining dependable boundaries between threads and their states is so difficult, ensuring that a library (what for Node would be a "module") is thread-safe consumes a great deal of developer time. Can I know that this library will not destroy some part of my application? Guaranteeing thread safety requires great diligence on the part of a library's developer and these guarantees may be conditional: for example, a library may be thread-safe when reading—but not when writing.

The primary argument for single-threading is that control flow is difficult in concurrent environments, and especially so when memory access or code execution order is unpredictable:

- Instead of concerning themselves with arbitrary locking and other collisions, developers can focus on constructing execution chains whose ordering is predictable.
- Because parallelization is accomplished through the use of multiple processes, each with an individual and distinct memory space, communication between processes remains uncomplicated—via the Rule of Simplicity we achieve not only simple and bug-free components, but easier interoperability as well.
- Because state is not (arbitrarily) shared between individual Node processes, a single process is automatically protected from surprise visits from other processes bent on memory reallocation or resource monopolization. Communication is through clear channels using basic protocols, all of which makes it very hard to write programs that make unpredictable changes across processes.
- Thread-safety is one less concern for developers to waste time worrying about. Because single-threaded concurrency obviates the collisions present in multithreaded concurrency, development can proceed more quickly, on surer ground.



A single thread efficiently managed by an event loop brings stability, maintainability, readability, and resilience to Node programs. The big news is that Node continues to deliver the speed and power of multithreading to its developers—the brilliance of Node's design makes such power transparent, reflecting one part of Node's stated aim of bringing the most power to the most people with the least difficulty.



In the preceding diagram, the differences between two single-threaded models and a multithreaded model are shown.

There is no escape from blocking operations—reading from a file, for example, will always take some time. A single-threaded synchronous model forces each task to wait for others to finish prior to starting, consuming more time. Several tasks can be started in parallel using threads, even at different times, where total execution time is no longer than that taken by the longest running thread. When using threads, the developer becomes responsible for synchronizing the activity of each individual thread, using locks or other scheduling tools. This can become very complex when the number of threads increases, and in this complexity lives very subtle and hard-to-find bugs.

Rather than having the developer struggle with this complexity, Node itself manages I/O threads. You need not micromanage I/O threading; one simply designs an application to establish data availability points (callbacks) and the instructions to be executed once the said data is available. Threads provide the same efficiency under the hood, yet their management is exposed to the developer through an easily comprehensible interface.

Multithreading is already native and transparent

Node's I/O thread pool executes within the OS scope, and its work is distributed across cores (just as any other job scheduled by the OS would be similarly distributed). When you are running Node, you are already taking advantage of its multithreaded execution.

In the upcoming discussion of child processes and the `Cluster` module, we will see this style of parallelism—of multiple parallel processes—in action. We will see how Node is not denied the full power of an OS.

As we saw earlier, when discussing Node's core architecture, the V8 thread in which one executes JavaScript programs is bound to **libuv**, which functions as the main, system-level, I/O event dispatcher. In this capacity, `libuv` handles the timers, filesystem calls, network calls, and other I/O operations requested by the relevant JavaScript process or module commands, such as `fs.readFile`, `http.createServer`, and so on. Therefore, the main V8 event loop is best understood as a control-flow programming interface, supported and powered by the highly-efficient, multithreaded, system delegate `libuv`.

Burt Belder, one of Node's core contributors, is also one of the core contributors to `libuv`. In fact, Node's development has provoked a simultaneous increase in `libuv` development, a feedback loop that has only improved the speed and stability of both projects. It has merged and replaced the `libeo` and `libev` libraries that formed the original core of Node's stack.

Consider another of Raymond's rules, the **Rule of Separation**: "Separate policy from mechanism; separate interfaces from engines". The engine that powers Node's asynchronous, event-driven style of programming is `libuv`; the interface to that engine is V8's JavaScript runtime. Continuing with Raymond:

> *One way to effect that separation is, for example, to write your application as a library of C service routines that are driven by an embedded scripting language, with the application flow of control written in the scripting language rather than C.*

The ability to orchestrate hyper-efficient parallel OS processes within the abstraction of a single predictable thread exists by design, not as a concession. It concludes a pragmatic analysis of how the application development process can be improved, and it is certainly not a limitation on what is possible.

A detailed unpacking of `libuv` can be found at `http://nikhilm.github.io/uvbook/`. Burt Belder also gives an in-depth talk on how `libuv` works under the hood at `http://www.youtube.com/watch?v=nGn60vDSxQ4`.

Creating child processes

Software development is no longer the realm of monolithic programs. Applications running on networks cannot forego interoperability. Modern applications are distributed and decoupled. We now build applications that connect users with resources distributed across the Internet. Many users are accessing shared resources simultaneously. A complex system is easier to understand if the whole is understood as a collection of interfaces to programs that solve one or a few clearly defined, related problems. In such a system it is expected (and desirable) that processes do not sit idle.

An early criticism of Node was that it did not have multicore awareness. That is, if a Node server were running on a machine with several cores, it would not be able to take advantage of this extra horsepower. Within this seemingly reasonable criticism hid an unjustified bias based on a straw man: a program that is unable to explicitly allocate memory and execution "threads" in order to implement parallelization cannot handle enterprise-grade problems.

This criticism is a persistent one. It is also not true.

While a single Node process runs on a single core, any number of Node processes can be "spun up" through use of the `child_process` module. Basic usage of this module is straightforward: we fetch a `ChildProcess` object, and listen for data events. This example will call the Unix command `ls`, listing the current directory:

```
var spawn = require('child_process').spawn;
var ls    = spawn('ls', ['-lh', '.']);
ls.stdout.on('readable', function() {
  var d = this.read();
  d && console.log(d.toString());
});
ls.on('close', function(code) {
  console.log('child process exited with code ' + code);
});
```

Here, we spawn the `ls` process (list directory), and read from the resulting readable `Stream`, receiving something like:

```
-rw-r--r-- 1 root root   43 Jul  9 19:44 index.html
-rw-rw-r-- 1 root root  278 Jul 15 16:36 child_example.js
-rw-r--r-- 1 root root 1.2K Jul 14 19:08 server.js

child process exited with code 0
```

Any number of child processes can be spawned in this way. It is important to note here that when a child process is spawned, or otherwise created, the OS itself assigns the responsibility for that process to a given CPU. Node is not responsible for how an OS allocates resources. The upshot is that on a machine with eight cores it is likely that spawning eight processes will result in each being allocated to independent processors. In other words, child processes are automatically spread by the OS across CPUs, putting the lie to claims that Node cannot take full advantage of multicore environments.

Each new Node process (child) is allocated 10 MB of memory, and represents a new V8 instance that will take at least 30 milliseconds to start up. While it is unlikely that you will be spawning many thousands of these processes, understanding how to query and set OS limits on user-created processes is beneficial. `htop` or `top` will report the number of processes currently running, or you can use `ps aux | wc -l` from the command line. The Unix command `ulimit` (`http://ss64.com/bash/ulimit.html`) provides important information on user limits on an OS. Passing `ulimit`, the `-u` argument will show the maximum number of user processes that can be spawned. Changing the limit is accomplished by passing it as an argument: `ulimit -u 8192`.

The `child_process` module represents a class exposing four main methods: `spawn`, `fork`, `exec`, and `execFile`. These methods return a `ChildProcess` object that extends `EventEmitter`, exposing an interface to child events and a few functions that are helpful to managing child processes. We'll take a look at its main methods, and follow up with a discussion of the common `ChildProcess` interface.

Spawning processes

This powerful command allows a Node program to start and interact with processes spawned via system commands. In the preceding example, we used `spawn` to call a native OS process, `ls`, passing that command the `lh` and `.` arguments. In this way, any process can be started just as one might start it via a command line. The method takes three arguments:

- **command**: A command to be executed by the OS shell
- **arguments** (optional): These are command-line arguments, sent as an array
- **options**: An optional map of settings for `spawn`

The options for `spawn` allow its behavior to be carefully customized:

- `cwd` (String): By default, the command will understand its current working directory to be the same as that of the Node process calling `spawn`. Change that setting using this directive.
- `env` (Object): This is used to pass environment variables to a child process. For instance, consider spawning a child with an environment object such as the following:

  ```
  {
    name : "Sandro",
    role : "admin"
  }
  ```

  The child process environment will have access to these values.

- `detached` (Boolean): When a parent spawns a child, both processes form a group, and the parent is normally the leader of that group. To make a child the group leader, use `detached`. This will allow the child to continue running even after the parent exits. This is because the parent will wait for the child to exit by default. You can call `child.unref()` to tell the parent's event loop that it should not count the child reference, and exit if no other work exists.
- `uid` (Number): Set the `uid` (user identity) directive for the child process, in terms of standard system permissions, such as a UID that has execute privileges on the child process.
- `gid` (Number): Set the `gid` (group identity) directive for the child process, in terms of standard system permissions, such as a GID that has execute privileges on the child process.

- `stdio` (String or Array): Child processes have file descriptors, the first three being the standard I/O descriptors `process.stdin`, `process.stdout` and `process.stderr`, in order (fds = 0,1,2). This directive allows those descriptors to be redefined, inherited, and so forth.

  Consider the output of the following child process program:

  ```
  process.stdout.write(new Buffer("Hello!"));
  ```

  Here, a parent would listen on `child.stdout`. If instead we want a child to inherit its parent's `stdio`, such that when the child writes to `process.stdout` what is emitted is piped through to the parent's `process.stdout`, we would pass the relevant parent file descriptors to the child, overriding its own:

  ```
  spawn("node", ['./reader.js', './afile.txt'], {
    stdio: [process.stdin, process.stdout, process.stderr]
  });
  ```

  In this case, the child's output would pipe straight through to the parent's standard output channel. Also, see `fork`, below, for more information on this kind of pattern.

  Each of the three (or more) file descriptors can take one of six values:

  - `pipe`: This creates a pipe between the child and the parent. As the first three child file descriptors are already exposed to the parent (`child.stdin`, `child.stdout`, and `child.stderr`) this is only necessary in more complex child implementations.
  - `ipc`: This creates an IPC channel for passing messages between a child and parent. A child process may have a maximum of one IPC file descriptor. Once this connection is established, the parent may communicate with the child via `child.send`. If the child sends JSON messages through this file descriptor, those emissions can be caught by using `child.on("message")`. If running a Node program as a child, it is likely a better choice to use `ChildProcess.fork`, which has this messaging channel built in.
  - `ignore`: The file descriptors 0-2 will have `/dev/null` attached to them. For others, the referenced file descriptor will not be set on the child.
  - **A stream object**: This allows the parent to share a stream with the child. For demonstration purposes, given a child that will write the same content to any provided `WritableStream`, we could do something like this:

  ```
  var writer = fs.createWriteStream("./a.out");
  writer.on('open', function() {
  ```

```
    var cp = spawn("node", ['./reader.js'], {
     stdio: [null, writer, null]
    });
});
```

The child will now fetch its content and pipe it to whichever output stream it has been sent:

```
fs.createReadStream('cached.data').pipe(process.stdout);
```

- **An integer**: A file descriptor id.
- `null` and `undefined`: These are the default values. For file descriptors 0-2 (`stdin`, `stdout`, and `stderr`) a pipe is created. Others default to `ignore`.

In addition to passing `stdio` settings as an array, certain common groupings can be implemented by passing a shortcut string value:

- `'ignore' = ['ignore', 'ignore', 'ignore']`
- `'pipe' = ['pipe', 'pipe', 'pipe']`
- `'inherit' = [process.stdin, process.stdout, process.stderr]` or `[0,1,2]`

We have shown some examples of using `spawn` to run Node programs as child processes. While this is a perfectly valid usage (and a good way to try out the API options), `spawn` is primarily for running system commands. See the discussion of `fork`, below, for more information on running Node processes as children.

It should be noted that the ability to spawn any system process means that one can use Node to run other application environments installed on the OS. If one had the popular PHP language installed, the following would be possible:

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

var php = spawn("php", ['-r', 'print "Hello from PHP!";']);

php.stdout.on('readable', function() {
  var d;
  while(d = this.read()) {
    console.log(d.toString());
  }
});

//  Hello from PHP!
```

Running a more interesting, larger program would be just as easy.

Apart from the ease with which one might run Java or Ruby or other programs through Node using this technique, asynchronously, we also have here a good answer to a persistent criticism of Node: JavaScript is not as fast as other languages for crunching numbers, or doing other CPU-heavy tasks. This is true, in the sense that Node is primarily optimized for I/O efficiency and helping with the management of high-concurrency applications, and JavaScript is an interpreted language without a strong focus on heavy computation.

However, using `spawn`, one can very easily pass off massive computations and long-running routines on analytics engines or calculation engines to separate processes in other environments. Node's simple event loop will be sure to notify the main application when those operations are done, seamlessly integrating the resultant data. In the meantime, the main application is free to keep serving clients.

Forking processes

Like `spawn`, `fork` starts a child process, but is designed for running Node programs with the added benefit of having a communication channel built in. Rather than passing a system command to `fork` as its first argument, one passes the path to a Node program. As with `spawn`, command-line options can be sent as a second argument, accessible via `process.argv` in the forked child process.

An optional options object can be passed as its third argument, with the following parameters:

- `cwd` (String): By default, the command will understand its current working directory to be the same as that of the Node process calling `fork`. Change that setting using this directive.
- `env` (Object): This is used to pass environment variables to a child process. See `spawn`.
- `encoding` (String): This sets the encoding of the communication channel.
- `execPath` (String): This is the executable used to create the child process.
- `silent` (Boolean): By default, a forked child will have its `stdio` associated with the parent's (`child.stdout` is identical to `parent.stdout`, for example). Setting this option to `true` disables this behavior.

An important difference between `fork` and `spawn` is that the former's child process *does not automatically exit* when it is finished. Such a child must explicitly exit when it is done, easily accomplished via `process.exit()`.

In the following example, we create a child that emits an incrementing number every tenth of a second, which its parent then dumps to the system console. First, the child program:

```
var cnt = 0;

setInterval(function() {
  process.stdout.write(" -> " + cnt++);
}, 100);
```

Again, this will simply write a steadily increasing number. Remembering that with `fork` a child will inherit the `stdio` of its parent, we only need create the child in order to get output in a terminal running the parent process:

```
var fork = require('child_process').fork;
fork('./emitter.js');

// -> 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 ...
```

The `silent` option can be demonstrated here. The following turns off any output to the terminal:

```
fork('./emitter.js', [], { silent: true });
```

Creating multiple, parallel processes is easy. Let's multiply the number of children created:

```
fork('./emitter.js');
fork('./emitter.js');
fork('./emitter.js');

// 0 -> 0 -> 0 -> 1 -> 1 -> 1 -> 2 -> 2 -> 2 -> 3 -> 3 -> 3 -> 4 ...
```

It should be clear at this point that by using `fork` we are creating many parallel execution contexts, spread across all machine cores.

This is straightforward enough, but the built-in communication channel `fork` provides makes communicating with forked children even easier, and cleaner. Consider the following:

- **Parent**

```
var fork = require('child_process').fork;
var cp = fork('./child.js');
cp.on('message', function(msgobj) {
  console.log('Parent got message:', msgobj.text);
});
```

```
cp.send({
  text: "I love you"
});
```

- **Child**

```
process.on('message', function(msgobj) {
  console.log('Child got message:', msgobj.text);
  process.send({
    text: msgobj.text + ' too'
  });
});
```

By executing the parent script, we will see the following in our console:

```
Child got message: I love you
Parent got message: I love you too
```

Buffering process output

In cases where the complete buffered output of a child process is sufficient, with no need to manage data through events, `child_process` offers the `exec` method. The method takes three arguments:

- **command**: A command-line string. Unlike `spawn` and `fork`, which pass arguments to a command via an array, this first argument accepts a full command string, such as `ps aux | grep node`.
- **options**: This is an optional argument.
  - `cwd` (String): This sets the working directory for the command process.
  - `env` (Object): This is a map of key-value pairs that will be exposed to the child process.
  - `encoding` (String): This is the encoding of the child's data stream. The default value is `'utf8'`.
  - `timeout` (Number): This specifies the milliseconds to wait for the process to complete, at which point the child process will be sent the `killSignal.maxBuffer` value.
  - `killSignal.maxBuffer` (Number): This is the maximum number of bytes allowed on `stdout` or `stderr`. When this number is exceeded, the process is killed. This default is 200 KB.

    - `killSignal` (String): The child process receives this signal after a timeout. This default is `SIGTERM`.

- **callback**: This receives three arguments: an `Error` object, if any; `stdout` (a `Buffer` object containing the result); `stderr` (a `Buffer` object containing error data, if any). If the process was killed, `Error.signal` will contain the kill signal.

When you want the buffering behavior of `exec` but are targeting a Node file, use `execFile`. Importantly, `execFile` does not spawn a new subshell, which makes it slightly less expensive to run.

Communicating with your child

All instances of the `ChildProcess` object extend `EventEmitter`, exposing events useful for managing child data connections. Additionally, `ChildProcess` objects expose some useful methods for interacting with children directly. Let's go through those now, beginning with attributes and methods:

- `child.connected`: When a child is disconnected from its parent via `child.disconnect()`, this flag will be set to `false`.
- `child.stdin`: This is a `WritableStream` corresponding to the child's standard in.
- `child.stdout`: This is a `ReadableStream` corresponding to the child's standard out.
- `child.stderr`: This is a `ReadableStream` corresponding to the child's standard error.
- `child.pid`: This is an integer representing the process ID (PID) assigned to the child process.
- `child.kill`: This tries to terminate a child process, sending it an optional signal. If no signal is specified, the default is `SIGTERM` (for more about signals, see `http://unixhelp.ed.ac.uk/CGI/man-cgi?signal+7`). While the method name sounds terminal, it is not guaranteed to kill a process—it only sends a signal to a process. Dangerously, if `kill` is attempted on a process that has already exited, it is possible that another process that has been newly assigned the PID of the dead process will receive the signal, with indeterminable consequences. This method should fire a `close` event, which the signal used to close the process.

- `child.disconnect()`: This command severs the IPC connection between the child and its parent. The child will then die gracefully, as it has no IPC channel to keep it alive. You may also call `process.disconnect()` from within the child itself. Once a child has disconnected, the `connected` flag on that child reference will be set to `false`.

Sending messages to children

As we saw in our discussion of `fork`, and when using the `ipc` option on `spawn`, child processes can be sent messages via `child.send`, with the message passed as the first argument. A TCP server, or socket handle, can be passed along with the message as a second argument. In this way, a TCP server can spread requests across multiple child processes. For example, the following server distributes socket handling across a number of child processes equaling the total number of CPUs available. Each forked child is given a unique ID, which it reports when started. Whenever the TCP server receives a socket, that socket is passed as a handle to a random child process. That child process then sends a unique response, demonstrating that socket handling is being distributed.

- **Parent**

```
var fork = require('child_process').fork;
var net = require('net');
var children = [];

require('os').cpus().forEach(function(f, idx) {
  children.push(fork("./child.js", [idx]));
});

net.createServer(function(socket) {
  var rand = Math.floor(Math.random() * children.length);
  children[rand].send(null, socket);
}).listen(8080);
```

- **Child**

```
var id = process.argv[2];
process.on('message', function(n, socket) {
  socket.write('child ' + id + ' was your server today.\r\n');
  socket.end();
});
```

Start the parent server in a terminal window. In another window, run `telnet 127.0.0.1 8080`. You should see something similar to the following output, with a random child ID being displayed on each connection (assuming there exist multiple cores):

```
Trying 127.0.0.1...
…
child 3 was your server today.
Connection closed by foreign host.
```

Parsing a file using multiple processes

One of the tasks many developers will take on is the building of a logfile processor. A logfile can be very large and many megabytes long. Any single program working on a very large file can easily run into memory problems or simply run much too slowly. It makes sense to process a large file in pieces. We're going to build a simple log processor that breaks up a big file into pieces and assigns one to each of several child workers, running them in parallel.

The entire code for this example can be found in the `logproc` folder of the code bundle. We will focus on the main routines:

- Determining the number of lines in the logfile
- Breaking those up into equal chunks
- Creating one child for each chunk and passing it parse instructions
- Assembling and displaying the results

To get the word count of our file, we use the `wc` command with `child.exec` as shown in the following code:

```
child.exec("wc -l " + filename, function(e, fL) {
  fileLength = parseInt(fL.replace(filename, ""));
  var fileRanges = [];
  var oStart = 1;
  var oEnd = fileChunkLength;

  while(oStart < fileLength) {
    fileRanges.push({
      offsetStart : oStart,
      offsetEnd   : oEnd
    })
```

```
    oStart = oEnd + 1;
    oEnd = Math.min(oStart + fileChunkLength, fileLength);
  }
```

Let's say we use `fileChunkLength` of 500,000 lines. This means four child processes are to be created, and each will be told to process a range of 500,000 lines in our file, such as 1 to 500,000:

```
var w = child.fork('bin/worker');
w.send({
  file          : filename,
  offsetStart   : range.offsetStart,
  offsetEnd     : range.offsetEnd
});
w.on('message', function(chunkData) {
  // pass results data on to a reducer.
});
```

Each of these workers will themselves use a child process to grab their allotted chunk, employing `sed`, the native **Stream Editor** for Unix:

```
process.on('message', function(m) {
  var filename = m.file;

    var sed = "sed -n '" + m.offsetStart + "," + m.offsetEnd + "p' " +
filename;

    var reader = require('child_process').exec(sed, {
      maxBuffer : 1024 * 1000000
    }, function(err, data, stderr) {

      //  Split the file chunk into lines and process it.
      //
        data = data.split("\n");
        ...
```

Here we are executing the command `sed -n '500001,1000001p' logfile.txt`, which plucks the given range of lines and returns them for processing. Once we're done processing the columns of data (adding them up, and so forth), this child will return its data to the master (as previously described) and the data results will be written to a file, otherwise manipulated, or sent to `stdout`, as shown in the following output:

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ FILE: ./short.log     CPUS: 8 +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++Stats++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Operation took: 1.076 seconds
Log start: December 28th 2011, 6:03:26 am
Log end: December 28th 2011, 6:28:29 am
Total Seconds: 1503.478
Total Datapoints: 2000000
Throughput: 1330.249/second
Outliers under (0): 0 (%0.000)
Outliers over (10): 1112268 (%55.613)
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++Distribution (Milliseconds : Count)++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ 0 :   0              (%0.000)
+ 1 :   0              (%0.000)
+ 2 :   0              (%0.000)
+ 3 :   0              (%0.000)
+ 4 :   666            (%0.033)
+ 5 :   4520           (%0.226)
+ 6 :   18514          (%0.926)
+ 7 :   47180          (%2.359)
+ 8 :   90920          (%4.546)
+ 9 :   182262         (%9.113)
+ 10 :  543670         (%27.184)
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++Percentiles++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ 0 :   0.000          (100.000)
+ 1 :   0.000          (100.000)
+ 2 :   0.000          (100.000)
+ 3 :   0.000          (100.000)
+ 4 :   0.033          (99.967)
+ 5 :   0.259          (99.741)
+ 6 :   1.185          (98.815)
+ 7 :   3.544          (96.456)
+ 8 :   8.090          (91.910)
+ 9 :   17.203         (82.797)
+ 10 :  44.387         (55.613)
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

The full file for this example is much longer, but all of that extra code is merely formatting and other detail—the Node child process management we have described suffices to create a parallelized system for number crunching that will process many millions of lines of code in seconds. By using more processes, spread across more cores, the log parsing speed can be reduced even further.

Once you have the `logproc` folder, run the command `npm install`, and execute an example log parse with the command `node bin/master.js -f ./short.log -rmin 0 -rmax 20`. You should see a simple distribution mapping of the data contained in `short.log`. Go ahead and play with the numbers.

Using the cluster module

As we saw when processing large logfiles, the pattern of a master parent controller for many child processes is just right for vertical scaling in Node. In response to this, the Node API has been augmented by a `cluster` module, which formalizes this pattern and helps to make its achievement easier. Continuing with Node's core purpose of helping to make scalable network software easier to build, the particular goal of cluster is to facilitate the sharing of network ports amongst many children.

For example, the following code creates a cluster of worker processes all sharing the same HTTP connection:

```
var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;

if(cluster.isMaster) {
  for(var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
}

if(cluster.isWorker) {
  http.createServer(function(req, res) {
    res.writeHead(200);
    res.end("Hello from " + cluster.worker.id);
  }).listen(8080);
}
```

We'll dig into the details shortly. For now, notice that `cluster.fork` has taken zero arguments. What does `fork` without a command or file argument do? Within a cluster, the default action is to fork the current program. We see during `cluster.isMaster`, the action is to fork children (one for each available CPU). When this program is re-executed in a forking context, `cluster.isWorker` will be `true` and a new HTTP server *running on a shared port* is started. Multiple processes are sharing the load for a single server.

Start and connect to this server with a browser. You will see something like `Hello from 8`, the integer corresponding to the unique `cluster.worker.id` value of the worker that assigned responsibility for handling your request. Balancing across all workers is handled automatically, such that refreshing your browser a few times will result in different worker IDs being displayed.

Later on, we'll go through an example of sharing a socket server across a cluster. For now, we'll lay out the cluster API, which breaks down into two sections: the methods, attributes, and events available to the cluster master, and those available to the child. As workers in this context are defined using fork, the documentation for that method of `child_process` can applied here as well.

- `cluster.isMaster`: This is the Boolean value indicating whether the process is a master.
- `cluster.isWorker`: This is the Boolean value indicating whether the process was forked from a master.
- `cluster.worker`: This will bear a reference to the current worker object, only available to a child process.
- `cluster.workers`: This is a hash containing references to all active worker objects, keyed by the worker ID. Use this to loop through all worker objects. This only exists within the master process.
- `cluster.setupMaster([settings])`: This is a convenient way of passing a map of default arguments to be used when a child is forked. If all children are going to fork the same file (as is often the case), you will save time by setting it here. The available defaults are as follows:
    - `exec` (String): This is the file path to the process file, defaulting to `__filename`.
    - `args` (Array): This contains Strings sent as arguments to the child process. The default is to fetch arguments with `process.argv.slice(2)`.
    - `silent` (Boolean): This specifies whether or not to send output to the master's `stdio`, defaulting to `false`.
- `cluster.fork([env])`: This creates a new worker process. Only the master process may call this method. To expose a map of key-value pairs to the child's process environment, send an object to `env`.
- `cluster.disconnect([callback])`: This is used to terminate all workers in a cluster. Once all the workers have died gracefully, the cluster process will itself terminate if it has no further events to wait on. To be notified when all children have expired, pass `callback`.

Cluster events

The cluster object emits several events listed as follows:

- `fork`: This is fired when the master tries to fork a new child. This is not the same as `online`. This receives a worker object.
- `online`: This is fired when the master receives notification that a child is fully bound. This differs from the `fork` event and receives a worker object.
- `listening`: When the worker performs an action that requires a `listen()` call (such as starting an HTTP server), this event will be fired in the master. The event emits two arguments: a worker object, and the address object containing the `address`, `port`, and `addressType` values of the connection.
- `disconnect`: This is called whenever a child disconnects, which can happen either through process exit events or after calling `child.kill()`. This will fire prior to the `exit` event—they are not the same. This receives a worker object.
- `exit`: Whenever a child dies this event is emitted. The event receives three arguments: a worker object, the exit code number, and the signal string, such as `SIGNUP`, which caused the process to be killed.
- `setup`: This is called after `cluster.setupMaster` has executed.

Worker object properties

Workers have the following attributes and methods:

- `worker.id`: This is the unique ID assigned to a worker, also representing the worker's key in the `cluster.workers` index.
- `worker.process`: This specifies a `ChildProcess` object referencing a worker.
- `worker.suicide`: The workers that have recently had `kill` or `disconnect` called on them will have their `suicide` attribute set to `true`.
- `worker.send(message, [sendHandle])`: Refer to `child_process.fork()`, which is previously mentioned.
- `worker.kill([signal])`: This kills a worker. The master can check this worker's `suicide` property in order to determine if the death was intentional or accidental. The default `signal` value that is sent is `SIGTERM`.
- `worker.disconnect()`: This instructs a worker to disconnect. Importantly, existing connections to the worker are not immediately terminated (as with `kill`), but are allowed to exit normally prior to the worker fully disconnecting. This is because existing connections may stay in existence for a very long time. It is a good pattern to regularly check if the worker has actually disconnected, perhaps using timeouts.

Worker events

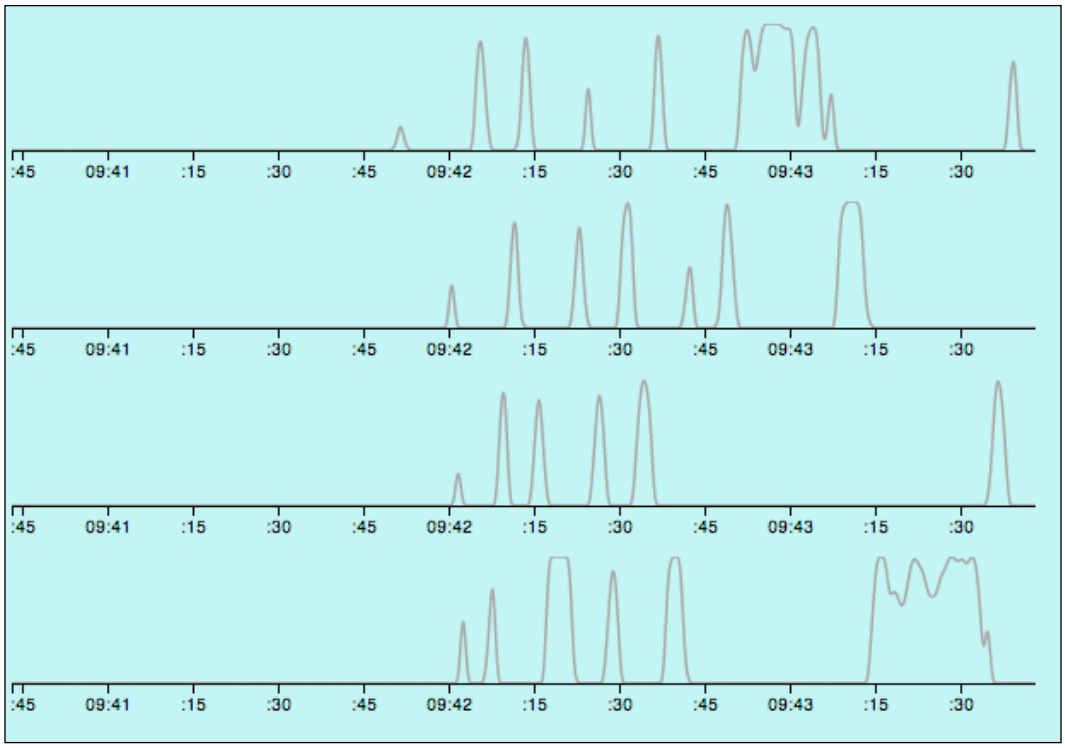
Workers also emit events, such as the ones mentioned in the following list:

- `message`: See `child_process.fork`
- `online`: This is identical to `cluster.online`, except that the check is against only the specified worker
- `listening`: This is identical to `cluster.listening`, except that the check is against only the specified worker
- `disconnect`: This is identical to `cluster.disconnect`, except that the check is against only the specified worker
- `exit`: See the `exit` event for `child_process`
- `setup`: This is called after `cluster.setupMaster` has executed

Now, using what we now know about the cluster module, let's implement a real-time tool for analyzing the streams of data emitted by many users simultaneously interacting with an application.

Real-time activity updates of multiple worker results

Using what we've learned we are going to construct a multiprocess system to track the behavior of all visitors to a sample web page. This will be composed of two main segments: a WebSocket-powered client library, which will broadcast each time a user moves a mouse, and an administration interface visualizing user interaction as well as when a user connects and disconnects from the system. Our goal is to show how a more complex system might be designed (such as one that tracks and graphs every click, swipe, or other interaction a user might make). The final administration interface will show activity graphs for several users and resemble this:



Because this system will be tracking the X and Y positions of each mouse motion made by all users, we will spread this continuous stream of data across all available machine cores using `cluster`, with each worker in the cluster sharing the burden of carrying the large amounts of socket data being fed into a single, shared port.

A good place to start is in designing the mock client page, which is responsible solely for catching all mouse movement events and broadcasting them, through a WebSocket, to our clustered socket server. We are using the native `WebSocket` implementation; you may want to use a library to handle older browsers (such as Socket.IO):

```
<head>
  <script>
    var connection = new WebSocket('ws://127.0.0.1:8081', ['json']);
    connection.onopen = function() {
      var userId = 'user' + Math.floor(Math.random()*10e10);
      document.onmousemove = function(e) {
        connection.send(JSON.stringify({
          id : userId,
          x  : e.x,
```

```
            y  : e.y
          }));
        }
      };
    </script>
  </head>
```

Here, we need to simply turn on the basic `mousemove` tracking, which will broadcast the position of a user's mouse on each movement to our socket. Additionally, we send along a unique user ID, as a tracking client identity will be important to us later on. Note that in a production environment you will want to implement a more intelligent unique ID generator, likely though a server-side authentication module.

In order for this information to reach other clients, a centralized socket server must be set up. As mentioned, we will want this socket server to be clustered. Clustered child processes, each duplicates of the following program, will handle mouse data sent by clients:

```
var SServer = require('ws').Server;
var socketServer = new SServer({
port: 8081
});
socketServer.on('connection', function(socket) {
  var lastMessage = null;
  var kill = function() {
      if(lastMessage) {
process.send({
      kill : lastMessage.id
      });
    }
  };
    socket.on('message', function(message) {
      lastMessage = JSON.parse(message);
      process.send(lastMessage);
    });
    socket.on('close', kill);
    socket.on('error', kill);
});
```

In this demonstration, we are using Einar Otto Stangvik's very fast and well-designed socket server library, `ws`, which is hosted on GitHub at `https://github.com/einaros/ws`.

Thankfully our code remains very simple. We have a socket server listening for messages (remember that the client is sending an object with mouse X and Y, as well as a user ID). Finally, when data is received (the `message` event), we parse the received JSON into an object and pass that back to our cluster master via `process.send`. Note as well how we store the last message (`lastMessage`), done for bookkeeping reasons, as when a connection terminates we will need to pass along the last user ID seen on this connection to administrators.

The pieces to catch client data broadcasts are now set up. Once this data is received, how is it passed to the administration interface previously pictured?

We've designed this system with scaling in mind, and we want to decouple the collection of data from the systems that broadcast data. Our cluster of socket servers can accept a constant flow of data from many thousands of clients, and should be optimized for doing just that. In other words, the cluster should delegate the responsibility for broadcasting mouse activity data to another system, even to other servers.

In the next chapter we will look at more advanced scaling and messaging tools, such as message queues and UDP broadcasting. For our purposes here, we will simply create an HTTP server responsible for managing connections from administrators, and broadcasting mouse activity updates to them. We will use SSE for this, as the data flow need only be one-way, from server to client.

The HTTP server will implement a very basic validation system for administrator logins, holding on to successful connections in a way that will allow our socket cluster to broadcast mouse activity updates to all. It will also serve as a basic static file server, sending both the client and administration HTML when requested, though we will focus only on how it handles two routes: `admin/adminname`; and `/receive/adminname`. Once the server is understood, we will then go into how our socket cluster connects to it.

The first route `/admin/adminname` is mostly responsible for validating administrator login, also ensuring that this is not a duplicate login. Once that identity is established, we can send back an HTML page to the administration interface. The specific client code used to draw the graphs previously pictured won't be discussed here. What we do need is an SSE connection to our server such that the interface's graphing tools receive real-time updates of mouse activity. Some JavaScript on the returned administrator's page establishes such a connection:

```
var ev = new EventSource('/receive/adminname');
ev.addEventListener("open", function() {
  console.log("Connection opened");
});
```

```
ev.addEventListener("message", function(data) {
  //  Do something with mouse data, like graph it.
}
```

On our server we implement the `/receive/adminname` route:

```
if(method === "receive") {
  //  Unknown admin; reject
  if(!admins[adminId]) {
    return response.end();
  }
  response.writeHead(200, {
    "Content-Type": "text/event-stream",
    "Cache-Control": "no-cache",
    "Connection": "keep-alive"
  });
  response.write(":" + Array(2049).join(" ") + "\n");
  response.write("retry: 2000\n");
  response.on("close", function() {
    admins[adminId] = {};
  });
  setInterval(function() {
    response.write("data: PING\n\n");
  }, 15000);

  admins[adminId].socket = response;
  return;
}
```

The main purpose of this route is to establish an SSE connection and to store the administrator's connection, such that we can later broadcast to it.

We will now add the pieces that will pass mouse activity data along to a visualization interface. Scaling this subsystem across cores using the `cluster` module is our next step. The cluster master now simply needs to wait for mouse data from its socket-serving children, as previously described.

We will use the same ideas presented in the earlier discussion of cluster, simply forking the preceding socket server code across all available CPUs:

```
if(cluster.isMaster) {
  var i;
  for(i=0; i < numCPUs; i++) {
    cluster.fork();
  }
  cluster
  .on('exit', function(worker, code, signal) {
    console.log('worker ' + worker.process.pid + ' died');
  })
  //  ...adding other listeners as needed

  //  Set up socket worker listeners
  Object.keys(cluster.workers).forEach(function(id) {
    cluster.workers[id].on('message', function(msg) {
      var a;
      for(a in admins) {
        if(admins[a].socket) {
          admins[a].socket.write("data: " + JSON.stringify(msg) +
"\n\n");
        }
      }
    });
  });
}
```

Mouse activity data pipes into a cluster worker through a socket and is broadcasted via `process.send` to the cluster master previously described. On each worker message we run through all connected administrators and send mouse data to their visualization interfaces, using SSE. The administrators can now watch for the arrival and exit of clients, as well as their individual level of activity.

Summary

This is the first chapter where we've really begun to test Node's *scalability* goal. Having considered the various arguments for and against different ways of thinking about concurrency and parallelism, we arrived at an understanding of how Node has successfully maintained the advantages of threading and parallel processing while wrapping all of that complexity within a concurrency model that is both easy to reason about and robust.

Having gone deeper into how processes work, and in particular how child processes can communicate with each other, even spawn further children, we looked at two use cases. An example of how to combine native Unix command processes seamlessly with custom Node processes led us to a performant and straightforward technique for processing large files. The `cluster` module was then applied to the problem of how to share responsibility for handling a busy socket between multiple workers, this ability to share socket handles between processes demonstrating a powerful aspect of Node's design.

Having seen how Node applications might be scaled vertically, we can now look into horizontal scaling across many systems and servers. In the next chapter we'll learn how to connect Node with third-party services, such as Amazon and Facebook, communicate across networks with message queues, set up multiple Node servers behind proxies, and more.

8
Scaling Your Application

> *Evolution is a process of constant branching and expansion.*
>
> *– Stephen Jay Gould*

Scalability and performance are not the same thing:

> *The terms "performance" and "scalability" are commonly used interchangeably, but the two are distinct: performance measures the speed with which a single request can be executed, while scalability measures the ability of a request to maintain its performance under increasing load. For example, the performance of a request may be reported as generating a valid response within three seconds, but the scalability of the request measures the request's ability to maintain that three-second response time as the user load increases.*
>
> *– Steven Haines, "Pro Java EE 5"*

In the previous chapter we looked at how Node clusters might be used to increase the performance of an application. Through the use of clusters of processes and workers we've learned how to efficiently deliver results in the face of many simultaneous requests. We learned to scale Node *vertically*, keeping the same footprint (a single server) and increasing throughput by piling on the power of available CPUs.

In this chapter we will focus on *horizontal* scalability: the idea is that an application composed of self-sufficient and independent units (servers) can be scaled by adding more units without altering the application's code.

We want to create an architecture within which any number of optimized and encapsulated Node-powered servers can be added or subtracted in response to changing demands, dynamically scaling without ever requiring a system rewrite. We want to share work across different systems, pushing requests to the OS, to another server, to a third-party service, while coordinating those I/O operations intelligently using Node's evented approach to concurrency.

Through architectural parallelism, our systems can manage increased data volume more efficiently. Specialized systems can be isolated when necessary, even independently scaled or otherwise clustered.

Node is particularly well suited to handle two key aspects of horizontally scaled architectures.

First, Node enforces non-blocking I/O, such that the seizing up of any one unit will not cause a cascade of locking that brings down an entire application. Because no single I/O operation will block the entire system, integrating third-party services can be done with confidence, encouraging a decoupled architecture.

Second, Node places great importance on supporting as many fast network communication protocols as possible. Whether through a shared database, a shared filesystem, or a message queue, Node's efficient network and `Stream` layers allow many servers to synchronize their efforts in balancing load. Being able to efficiently manage shared socket connections, for instance, helps when scaling out a cluster of servers as much as it does a cluster of processes.

In this chapter we will look at how to balance traffic between many servers running Node, how these distinct servers can communicate, and how these clusters can bind to and benefit from specialized cloud services.

When to scale?

The theory around application scaling is a complex and interesting topic that continues to be refined and expanded. A comprehensive discussion of the topic would require several books, curated for different environments and needs. For our purposes we will simply learn how to recognize when scaling up (or even scaling down) is necessary.

Having a flexible architecture that can add and subtract resources as needed is essential to a resilient scaling strategy. A vertical scaling solution does not always always suffice (simply adding memory or CPUs will not deliver the necessary improvements). When should horizontal scaling be considered?

One simple but useful way to check the CPU and memory usage commanded by Node processes running on a server is to use the Unix `ps` command, in combination with `grep`:

```
ps –A u | grep node
```

This will list all of your Node processes, and their resource allotments.

Network latency

When network response times are exceeding some threshold, such as each request taking several seconds, it is likely that the system has gone well past a stable state.

While the easiest way to discover this problem is to wait for customer complaints about slow websites, it is better to create controlled stress tests against an equivalent application environment or server.

AB (**Apache Bench**) is a simple and straightforward way to do blunt stress tests against a server. This tool can be configured in many ways, but the kind of test you would do for measuring the network response times for your server is generally straightforward.

For example, let's test the response times for this simple Node server:

```
http.createServer(function(request, response) {
    response.writeHeader(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
}).listen(2112)
```

Here's how one might test running 10000 requests against that server, with a concurrency of 100 (the number of simultaneous requests):

```
ab -n 10000 -c 100 http://yourserver.com/
```

If all goes well, you will receive a report similar to this:

```
Concurrency Level:      100
Time taken for tests:   9.658 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      1120000 bytes
HTML transferred:       110000 bytes
Requests per second:    1035.42 [#/sec] (mean)
```

```
Time per request:       96.579 [ms] (mean)
Time per request:       0.966 [ms] (mean, across all concurrent requests)
Transfer rate:          113.25 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.4      0       6
Processing:    54   96  11.7     90     136
Waiting:       53   96  11.7     89     136
Total:         54   96  11.6     90     136

Percentage of the requests served within a certain time (ms)
  50%     90
  66%     98
      …
  99%    133
 100%    136 (longest request)
```

There is a lot of useful information contained in this report. In particular one should be looking for failed requests and the percentage of long running requests.

Much more sophisticated testing systems exist, but **ab** is a good quick-and-dirty snapshot of performance. Get in the habit of creating testing environments that mirror your production systems, and testing them.

Running ab on the same server running the Node process you are testing will of course impact the test speeds. The test runner itself uses a lot of server resources, so your results will be misleading. Full documentation for ab can be found in the following link:

`http://httpd.apache.org/docs/2.2/programs/ab.html`

Hot CPUs

When CPU usage begins to nudge maximums, start to think about increasing the number of units processing client requests. Remember that while adding one new CPU to a single-CPU machine will bring immediate and enormous improvements, adding another CPU to a 32-core machine will not necessarily bring an equal improvement. Slowdowns are not always about slow calculations.

As mentioned earlier, **htop** is a great way to get a quick overview of your server's performance. Because it visualizes the load being put on each core in real time, it is a great way to get an idea of what is happening. Additionally, the load average of your server is nicely summarized with three values. This is a happy server:

Load average: 0.00 0.01 0.00

What do these values mean? What is a "good" or a "bad" load average?

All three numbers are measuring CPU usage, presenting measurements taken at one, five, and fifteen-minute intervals. Generally it can be expected that short-term load will be higher than long-term load. If on average over time your server is not overly stressed, it is likely that clients are having a good experience.

On a single-core machine, load average should remain between 0.00 and 1.00. Any request will take *some*. Time—the question is whether the request is taking *more time than necessary*— and whether there are delays due to excessive load.

If a CPU can be thought of as a pipe, a measurement of 0.00 means that there is no excessive friction, or delay, in pushing through a drop of water. A measurement of 1.00 indicates that our pipe is at its capacity—water is flowing smoothly, but any additional attempts to push water through will be faced with delays, or backpressure. This translates into latency on the network, with new requests joining an ever-growing queue.

A multicore machine simply multiplies the measurement boundary. A machine with four cores is at its capacity when load average reaches 4.00.

How you choose to react to load averages depends on the specifics of an application. It is not unusual for servers running mathematical models to see their CPU averages hit maximum capacity—in such cases you want *all* available resources dedicated to performing calculations. A file server running at capacity, on the other hand, is likely worth investigating.

Generally, a load average above 0.60 should be investigated. Things are not urgent, but there may be a problem around the corner. A server that regularly reaches 1.00 after all known optimizations have been made is a clear candidate for scaling, as of course is any server exceeding that average.

Node also offers native process information via the `os` module:

```
var os = require('os');
//    Load average, as an Array
console.log(os.loadavg());
//    Total and free memory
console.log(os.totalmem());
console.log(os.freemem());
//    Information about CPUs, as an Array
console.log(os.cpus());
```

Socket usage

When the number of persistent socket connections begins to grow past the capacity of any single Node server, however optimized, it will be necessary to think about spreading out the servers handling user sockets. Using `socket.io` it is possible to check the number of connected clients at any time using the following command:

```
io.sockets.clients()
```

In general it is best to track web socket connection counts within the application, via some sort of tracking/logging system.

Many file descriptors

When the number of file descriptors opened in an OS hovers close to its limit it is likely that an excessive number of Node processes are active, files are open, or other file descriptors (such as sockets or named pipes) are in play. If these high numbers are not due to bugs or a bad design, it is time to add a new server.

Checking the number of open file descriptors, of any kind, can be accomplished using `lsof`:

```
# lsof | wc -l     // 1345
```

Data creep

When the amount of data being managed by a single database server begins to exceed many millions of rows or very many gigabytes of memory, it is time to think about scaling. Here you might choose to simply dedicate a single server to your database, begin to shard databases, or even move into a managed cloud storage solution earlier rather than later. Recovering from a data layer failure is rarely a quick fix, and in general it is dangerous to have a single point of failure for something as important as *all of your data*.

If you're using Redis, the `info` command will provide most of the data you will need, to make these decisions. For example:

```
redis> info
# Clients
connected_clients:1
blocked_clients:0
# Memory
used_memory:17683488
used_memory_human:16.86M
used_memory_rss:165900288
used_memory_peak:226730192
used_memory_peak_human:216.23M
used_memory_lua:31744
mem_fragmentation_ratio:9.38
# CPU
used_cpu_sys:13998.77
used_cpu_user:21498.45
used_cpu_sys_children:1.60
used_cpu_user_children:7.19
...
```

More information on INFO can be found at this link:

`http://redis.io/commands/info`

For MongoDB you might use the `db.stats()` command:

```
> db.stats(1024)
{    "collections" : 3,
    "objects" : 5,
    "avgObjSize" : 39.2,
    "dataSize" : 0,
    "storageSize" : 12,
    "numExtents" : 3,
    "indexes" : 1,
    "indexSize" : 7,
    "fileSize" : 196608,
    "nsSizeMB" : 16,
    ...
    "ok" : 1 }
```

Passing the argument `1024` flags `stats` to display all values in kilobytes.
More information can be found here:

`http://docs.mongodb.org/manual/reference/command/dbStats/`

Tools for monitoring servers

There are several tools available for monitoring servers, but few designed specifically for Node. One strong candidate is **StrongOps** (`http://strongloop.com/strongloop-suite/strongops/`) from **StrongLoop**, a company founded by the key contributors of Node's core. This cloud service is easily integrated with a Node app, offering a useful dashboard visualizing CPU usage, average response times, and more.

Other good monitoring tools to consider are listed in the following:

- **Scout**: `https://scoutapp.com/`
- **Nagios**: `http://www.nagios.org/`
- **Munin**: `http://munin-monitoring.org/`
- **Monit**: `http://mmonit.com/monit/`

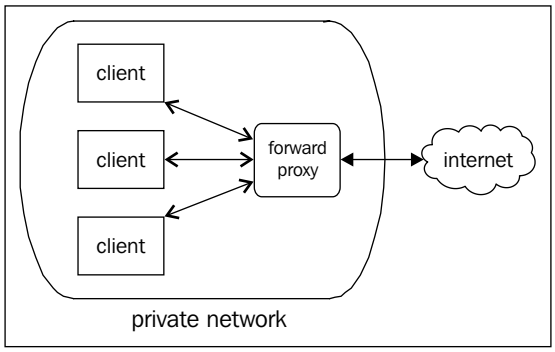
Running multiple Node servers

It is easy to purchase several servers and then to run some Node processes on them. But how can those distinct servers be coordinated such that they form part of a single application? One aspect of this problem concerns clustering multiple identical servers around a single entry point. How can client connections be shared across a pool of servers?

Forward and reverse proxies

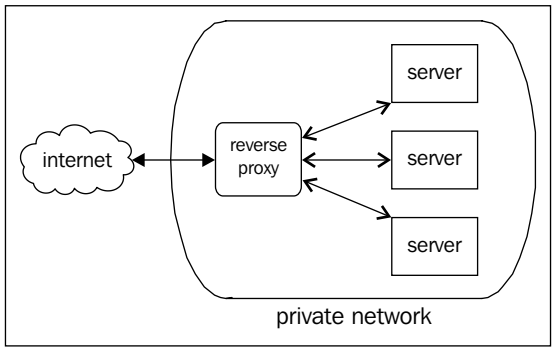
A **proxy** is someone or something acting on behalf of another.

A **forward proxy** normally works on behalf of clients in a private network, brokering requests to an outside network, such as retrieving data from the Internet. Earlier in this book we looked at how one might set up a proxy server using Node, where the Node server functioned as an intermediary, forwarding requests from clients to other network servers, usually via the Internet. Early web providers such as AOL functioned in the following way:



Network administrators use forward proxies when they must restrict access to the outside world, that is, the Internet. If network users are downloading malware from `somebadwebsite.com` via an e-mail attachment, the administrator can block access to that location. Restrictions on access to social networking sites might be imposed on an office network. Some countries even restrict access to public websites in this way.

A **reverse proxy**, not surprisingly, works in the opposite way, accepting requests from a public network and servicing those requests within a private network the client has little much visibility into. Direct access to servers by clients is first delegated to a reverse proxy:



This is the type of proxy we might use to balance requests from clients across many Node servers. Client *X* does not communicate with any given server directly. A broker *Y* is the point of first contact, able to direct *X* to a server that is under less load, or that is located closer to *X*, or is in some other way the best server for *X* to access at this time.

We will take a look at how to implement reverse proxies when scaling Node, discussing implementations that use **Nginx** and those using native Node modules.

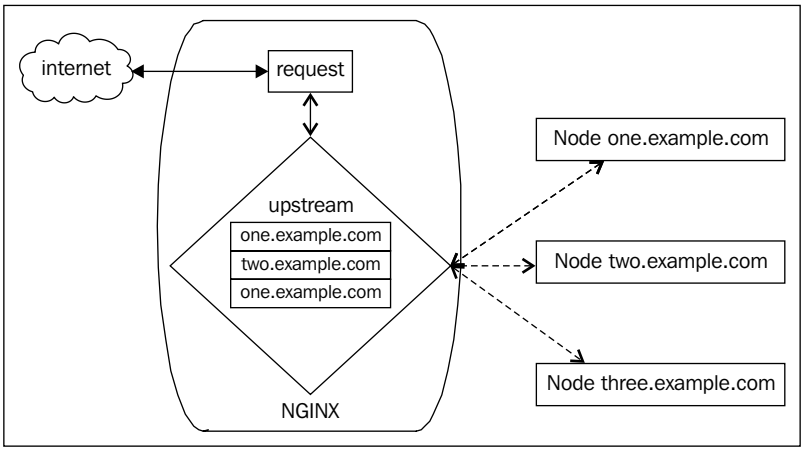
Nginx as a proxy

Nginx (pronounced as "Engine X") is a popular choice when hiding Node servers behind a proxy. Nginx is a very popular high-performance web server that is often used as a proxy server. There is some serendipity in the fact that Nginx is a popular choice with Node developers, given its design:

> *Nginx is able to serve more requests per second with [fewer] resources because of its architecture. It consists of a master process, which delegates work to one or more worker processes. Each worker handles multiple requests in an event-driven or asynchronous manner using special functionality from the Linux kernel (epoll/select/poll). This allows Nginx to handle a large number of concurrent requests quickly with very little overhead.*
>
> —`http://www.linuxjournal.com/magazine/nginx-high-performance-web-server-and-reverse-proxy`

Nginx also makes simple load balancing very easy. In our examples we will see how proxying through Nginx comes with load balancing "out of the box".

It is assumed that you have installed Nginx (if not, please visit `http://wiki.nginx.org/Main` to get started). We must now augment the `nginx.conf` file, in particular the `http` section. Our goal is to create a single Nginx server whose *sole* responsibility is to evenly distribute client requests across some number of Node servers, such that capacity for the entire system can be scaled without changing anything about how a client can or should connect to our application:



Within the `http` section of `nginx.config` are defined some **upstream** servers that will be candidates for redirection:

```
http {
...
    upstream app_myservers {
        server first.example.com;
        server second.example.com;
        server third.example.com;
    }
...
}
```

Now that we've established the candidate pool, we need to configure Nginx such that it will forward requests to one of its members:

```
server {
    listen       0.0.0.0:80;
    server_name  example.com my_website;
...
    location / {
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_set_header X-NginX-Proxy true;

        proxy_pass http://app_myservers/;
        proxy_redirect off;
    }
}
```

The key line is this one:

```
proxy_pass http://app_myservers/;
```

Note how the name `app_myservers` matches the name of our upstream definition. This should make what is happening clear: an Nginx server listening on port 80 will pass the request on to a server definition as contained in `app_myservers`. If the upstream definition has only one server in it, that server gets all the traffic. If several servers are defined, Nginx attempts to distribute traffic evenly among them.

It is also possible to balance load across several local servers using the same technique. One would simply run different Node servers on different ports, such as `server 127.0.0.1:8001; server 127.0.0.1:8002; ...`

Because we will likely want more precise control over how traffic is distributed across our upstream servers, there are further directives that can be applied to upstream server definitions.

In order to control the relative weighting of traffic distribution, we use the `weight` directive:

```
upstream app_myservers {
    server first.example.com weight=1;
    server second.example.com weight=2;
    server third.example.com weight=4;
}
```

This definition tells Nginx to distribute twice as much load to the second server as to the first, and four times as much to the third. For example, servers with more memory or CPUs might be favored. Another way to use this system is to create an **A/B** testing scenario, where one server containing a proposed new design receives some small fraction of the total traffic, such that metrics on the testing server (sales, downloads, engagement length, and so on) can be compared against the wider average.

Two other useful directives are available, which work together to manage connection failures:

- **max_fails**: This is the number of times communication with a server fails prior to marking that server as inoperative. The period of time within which these failures must occur is defined by `fail_timeout`.
- **fail_timeout**: The time slice during which `max_fails` must occur, indicating that a server is inoperative. This number also indicates the amount of time after a server is marked inoperative that Nginx will again attempt to reach the flagged server.

  For example:

```
upstream app_myservers {
    server first.myservers.com weight=1 max_fails=2 fail_
timeout=20s;
    server second.myservers.com weight=2 max_fails=10 fail_
timeout=5m;
    server third.myservers.com weight=4;
}
```

Full documentation for Nginx can be found at the following link:

`http://wiki.nginx.org/Configuration`

Using HTTP Proxy

Node is designed to facilitate the creation of network software, so it comes as no surprise that several proxying modules have been developed. The team at **NodeJitsu** has released the proxy they use in production, **HTTP proxy**. Let's take a look at how we would use it to route requests to different Node servers.

Unlike with Nginx, the entirety of our routing stack will exist in Node. One Node server will be running our proxy, listening on port 80. We'll cover the following three scenarios:

- Running multiple Node servers on separate ports on the same machine.
- Using one box as a pure router, proxying to external URLs.
- Creating a basic round-robin load balancer.

As an initial example, let's look at how to use this module to redirect requests:

```
var httpProxy = require('http-proxy');
var proxy = httpProxy.createServer({
target: {
  host: 'www.example.com',
  port: 80
}
}).listen(80);
```

By starting this server on port 80 of our local machine, we are able redirect the user to another URL.

To run several distinct Node servers, each responding to a different URL, on a single machine, one simply has to define a router:

```
var httpProxy = httpProxy.createServer({
    router: {
        'www.mywebsite.com'        : '127.0.0.1:8001',
        'www.myothersite.com'    : '127.0.0.1:8002',
    }
});
httpProxy.listen(80);
```

For each of your distinct websites you can now point your DNS name servers (via ANAME or CNAME records) to the same endpoint (wherever this Node program is running), and they will resolve to different Node servers. This is handy when you want to run several websites but don't want to create a new physical server for each one. Another strategy is to handle different paths within the same website on different Node servers:

```
var httpProxy = httpProxy.createServer({
    router: {
        'www.mywebsite.com/friends': '127.0.0.1:8001',
        'www.mywebsite.com/foes'    : '127.0.0.1:8002',
    }
});
httpProxy.listen(80);
```

This allows specialized functionality in your application to be handled by uniquely configured servers.

Setting up a load balancer is also straightforward. As with Nginx's **upstream** directive, we simply list the servers to be balanced and cycle through them:

```
var httpProxy = require('http-proxy');
var addresses = [
    { host: 'one.example.com', port: 80 },
    { host: 'two.example.com', port: 80 }
];
httpProxy.createServer(function(req, res, proxy) {
    var target = addresses.shift();
    proxy.proxyRequest(req, res, target);
    addresses.push(target);
}).listen(80);
```

Unlike with Nginx, we are responsible for doing the actual balancing. In this example, we treat servers equally, cycling through them in order. After the selected server is proxied, it is returned to the *rear* of the list.

It should be clear that this example could be easily extended to accommodate other directives, like Nginx's **weight**.

Another good option for proxying is the **bouncy** module created by *James Halliday*. You'll find the repository in the following link:

`https://github.com/substack/bouncy`

Message queues – RabbitMQ

One of the best ways to ensure that distributed servers maintain a dependable communication channel is to bundle the complexity of remote procedure calls into a distinct unit—a messaging queue. When one process wishes to send a message to another process, the message can simply be placed on this queue—like a to-do list for your application—with the queue service doing the work of ensuring messages get delivered as well as delivering any important replies back to the original sender.

There are a few enterprise-grade message queues available, many of them deploying AMQP **(Advanced Message Queueing Protocol)**. We will focus on a very stable and well-known implementation: RabbitMQ.

To install RabbitMQ in your environment follow the instructions found in the website: `http://www.rabbitmq.com/download.html`.

You will also need to install **Erlang** (instructions given under the **Official Clients** section at the preceding link).

Once installed you will start the RabbitMQ server with this command:

```
service rabbitmq-server start
```

To interact with RabbitMQ using Node we will use the node-amqp module created by *Theo Schlossnagle*:

```
npm install amqp
```

To use a message queue one must first create a consumer—a binding to RabbitMQ that will listen for messages published to the queue. The most basic consumer will listen for all messages:

```
var amqp = require('amqp');
var consumer = amqp.createConnection({ host: 'localhost', port: 5672
});
var exchange;
consumer.on('ready', function() {
    exchange = consumer.exchange('node-topic-exchange', {type:
      "topic"});
    consumer.queue('node-topic-queue', function(q) {
        q.bind(exchange, '#');
        q.subscribe(function(message) {
            //    Messages are buffers
            console.log(message.data.toString('utf8'));
        });
    });
});
```

We now are now listening for messages from the RabbitMQ server bound to port 5672.

Once this consumer establishes a connection, it will establish the name of the queue it will listen to, and should `bind` to an **exchange**. In this example we create a topic `exchange` (the default), giving it a unique name. We also indicate that we would like to listen for *all* messages via `#`. All that is left to do is subscribe to the queue, receiving a message object. We will learn more about the message object as we progress. For now, note the important `data` property, containing sent messages.

Now that we have established a consumer, let's publish a message to the exchange. If all goes well, we will see the sent message appear in our console:

```
consumer.on('ready', function() {
    // ...
exchange.publish("some-topic", "Hello!");
});
//    Hello!
```

We have already learned enough to implement useful scaling tools. If we have a number of distributed Node processes, even on different physical servers, each can reliably send messages to one another via RabbitMQ. Each process simply needs to implement an **exchange queue subscriber** to receive messages, and an **exchange publisher** to send messages.

Types of exchanges

RabbitMQ provides three types of exchanges: **direct**, **fanout**, and **topic**. The differences appear in the way each type of exchange processes **routing keys**—the first argument sent to `exchange.publish`.

A direct exchange matches routing keys directly. A queue binding like the following one matches *only* messages sent to `'room-1'`:

```
queue.bind(exchange, 'room-1');
```

Because no parsing is necessary, direct exchanges are able to process more messages than topic exchanges in a set period of time.

A fanout exchange is indiscriminate; it routes messages to all of the queues bound to it, ignoring routing keys. This type of exchange is used for wide broadcasts.

A topic exchange matches routing keys based on the wildcards **#** and *****. Unlike other types, routing keys for topic exchanges *must* be composed of words separated by dots—`"animals.dogs.poodle"`, for example. A `#` matches zero or more words—it will match every message (as we saw in the previous example), just like a fanout exchange. The other wildcard is *, and this matches *exactly* one word.

Direct and fanout exchanges can be implemented using nearly the same code as the given topic exchange example, requiring only that the exchange type be changed, and bind operations be aware of how they will be associated with routing keys (fanout subscribers receive all messages, regardless of the key; for direct the routing key must match directly).

This last example should drive home how topic exchanges work. We will create three queues with different matching rules, filtering the messages each queue receives from the exchange:

```
consumer.on('ready', function() {
    //    When all 3 queues are ready, publish.
    var cnt = 3;
    var queueReady = function() {
        if(--cnt > 0) {
            return;
        }
        exchange.publish("animals.dogs.poodles", "Poodle!");
        exchange.publish("animals.dogs.dachshund", "Dachshund!");
        exchange.publish("animals.cats.shorthaired", "Shorthaired
          Cat!");
        exchange.publish("animals.dogs.shorthaired", "Shorthaired
          Dog!");
        exchange.publish("animals.misc", "Misc!");
    }
    var exchange = consumer.exchange('topical', {type: "topic"});
    consumer.queue('queue-1', function(q) {
        q.bind(exchange, 'animals.*.shorthaired');
        q.subscribe(function(message) {
            console.log('animals.*.shorthaired -> ' +
              message.data.toString('utf8'));
        });
        queueReady();
    });
    consumer.queue('queue-2', function(q) {
        q.bind(exchange, '#');
        q.subscribe(function(message) {
            console.log('# -> ' + message.data.toString('utf8'));
        });
        queueReady();
    });
    consumer.queue('queue-3', function(q) {
        q.bind(exchange, '*.cats.*');
        q.subscribe(function(message) {
            console.log('*.cats.* -> ' +
              message.data.toString('utf8'));
        });
```

```
            queueReady();
        });
    });

    //    # -> Poodle!
    //    animals.*.shorthaired -> Shorthaired Cat!
    //    *.cats.* -> Shorthaired Cat!
    //    # -> Dachshund!
    //    # -> Shorthaired Cat!
    //    animals.*.shorthaired -> Shorthaired Dog!
    //    # -> Shorthaired Dog!
    //    # -> Misc!
```

The `node-amqp` module contains further methods for controlling connections, queues, and exchanges; in particular, it contains methods for removing queues from exchanges, and subscribers from queues. Generally, changing the makeup of a running queue on the fly can lead to unexpected errors, so use these with caution.

To learn more about the AMQP (and the options available when setting up with `node-amqp`) visit the following link:

`http://www.rabbitmq.com/tutorials/amqp-concepts.html`

Using Node's UDP module

UDP (User Datagram Protocol) is a lightweight core Internet messaging protocol, enabling servers to pass around concise **datagrams**. UDP was designed with a minimum of protocol overhead, forgoing delivery, ordering, and duplication prevention mechanisms in favor of ensuring high performance. UDP is a good choice when perfect reliability is not required and high-speed transmission is, such as what is found in networked video games and videoconferencing applications.

This is not to say that UDP is normally unreliable. In most applications it delivers messages with high probability. It is simply not suitable when *perfect* reliability is needed, such as in a banking application. It is an excellent candidate for monitoring and logging applications, and for non-critical messaging services.

Creating a UDP server with Node is straightforward:

```
var dgram = require("dgram");
var socket = dgram.createSocket("udp4");
socket.on("message", function(msg, info) {
    console.log("socket got: " + msg + " from " +
    info.address + ":" + info.port);
```

```
});
socket.bind(41234);
socket.on("listening", function() {
    console.log("Listening for datagrams.");
});
```

The `bind` command takes three arguments, which are as follows:

- **port**: The `Integer` port number.
- **address**: This is an optional address. If this is not specified, the OS will try to listen on all addresses (which is often what you want). You might also try using 0.0.0.0 explicitly.
- **callback**: This is an optional callback, which receives no arguments.

This socket will now emit a `message` event whenever it receives a datagram via the port 41234. The event callback receives the message itself as a first parameter, and a map of packet information as the second:

- **address**: The originating IP.
- **family**: One of IPv4 or IPv6.
- **port**: The originating port.
- **size**: The size of the message in bytes.

This map is similar to the map returned when calling `socket.address()`.

In addition to the message and listening events, a UDP socket also emits a **close** and **error** event, the latter receiving an `Error` object whenever an error occurs. To close a UDP socket (and trigger the close event), use `server.close()`.

Sending a message is even easier:

```
var client = dgram.createSocket("udp4");
var message = new Buffer("UDP says Hello!");
client.send(message, 0, message.length, 41234, "localhost",
function(err, bytes) {
    client.close();
});
```

The `send` method takes the form `client.send(buffer, offset, length, port, host, callback)`:

- `buffer`: A Buffer containing the datagram to be sent
- `offset`: An Integer indicating the position in buffer where the datagram begins

- `length`: The number of bytes in a datagram. In combination with **offset** this value identifies the full datagram within buffer
- `port`: An Integer identifying the destination port
- `address`: A String indicating the destination IP for the datagram
- `callback`: An optional callback function, called after the send has taken place

The size of a datagram cannot exceed 65507 bytes, which is equal to *2^16-1* (65535) bytes minus the 8 bytes used by the UDP header minus the 20 bytes used by the IP header.

We now have another candidate for inter-process messaging. It would be rather easy to set up a monitoring server for our node application, listening on a UDP socket for program updates and stats sent from other processes. The protocol speed is fast enough for real-time systems, and any packet loss or other UDP hiccups would be insignificant taken as a percentage of total volume over time.

Taking the idea of broadcasting further, we can also use the `dgram` module to create a multicast server. A **multicast** is simply a one-to-many server broadcast. We can broadcast to a range of IPs which have been permanently reserved as multicast addresses:

> *Host Extensions for IP Multicasting [RFC1112] specifies the extensions required of a host implementation of the Internet Protocol (IP) to support multicasting. The multicast addresses are in the range 224.0.0.0 through 239.255.255.255.*
>
> —`http://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml`

Additionally, the range between `224.0.0.0` and `224.0.0.255` is further reserved for special routing protocols.

As well, certain port numbers are allocated for use by UDP (and TCP), a list of which can be found in the following link:

`https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers`

The upshot of all this fascinating information is the knowledge that there is a block of IPs and ports reserved for UDP and/or multicasting, and we are now going to use some of them to implement multicasting over UDP with Node.

UDP multicasting with Node

The only difference between setting up a multicasting UDP server and a "standard" one is binding to a special UDP port for sending, and indicating that we'd like to listen to *all* available network adapters. Our multicasting server initialization looks like the following code snippet:

```
var socket = dgram.createSocket('udp4');
var multicastAddress     = '230.1.2.3';
var multicastPort     = 5554;
socket.bind(multicastPort);
socket.on("listening", function() {
    this.setMulticastTTL(64);
    this.addMembership(multicastAddress);
});
```

Once we've decided on a multicast port and an address and have bound, we catch the `listenng` event and configure our server. The most important command is `socket.addMembership`, which tells the kernel to join the multicast group at `multicastAddress`. Other UDP sockets can now subscribe to the multicast group at this address.

Datagrams hop through networks just like any network packet. The `setMulticastTTL` method is used to set the maximum number of hops (Time To Live) a datagram is allowed to make before it is abandoned, and not delivered. The acceptable range is 0-255, with the default being one (1) on most systems. This is not normally a setting one need worry about, but it is available when precise limits make sense, such as when packets are cheap and hops are costly.

If you'd like to also allow listening on the local interface, use `socket.setBroadcast(true)` and `socket.setMulticastLoopback(true)`. This is normally not necessary.

We are eventually going to use this server to broadcast messages to all UDP listeners on `multicastAddress`. For now, let's create two clients that will listen for multicasts:

```
dgram.createSocket('udp4')
.on('message', function(message, remote) {
    console.log('Client1 received message ' + message + ' from ' +
      remote.address + ':' + remote.port);
})
.bind(multicastPort, multicastAddress);
dgram.createSocket('udp4')
.on('message', function(message, remote) {
```

```
    console.log('Client2 received message ' + message + ' from ' +
      remote.address + ':' + remote.port);
})
.bind(multicastPort, multicastAddress);
```

We now have two clients listening to the same multicast port. All that is left to do is the multicasting. In this example we will use `setTimeout` to send a counter value every second:

```
var cnt = 1;
var sender;
(sender = function() {
    var msg = new Buffer("This is message #" + cnt);
    socket.send(
        msg,
        0,
        msg.length,
        multicastPort,
        multicastAddress
    );
    ++cnt;
    setTimeout(sender, 1000);
})();
```

The counter values will produce something like the following:

```
Client2 received message This is message #1 from 67.40.141.16:5554
Client1 received message This is message #1 from 67.40.141.16:5554
Client2 received message This is message #2 from 67.40.141.16:5554
Client1 received message This is message #2 from 67.40.141.16:5554
Client2 received message This is message #3 from 67.40.141.16:5554
...
```

We have two clients listening to broadcasts from a specific group. Let's add another client, listening on a different group—let's say at multicast address `230.3.2.1`:

```
dgram.createSocket('udp4')
.on('message', function(message, remote) {
    console.log('Client3 received message ' + message + ' from ' +
      remote.address + ':' + remote.port);
})
.bind(multicastPort, '230.3.2.1');
```

Because our server currently broadcasts messages to a different address, we will need to change our server configuration and add this new address with another `addMembership` call:

```
socket.on("listening", function() {
    this.addMembership(multicastAddress);
    this.addMembership('230.3.2.1');
});
```

We can now send to *both* addresses:

```
(sender = function() {
    socket.send(
        ...
        multicastAddress
    );
    socket.send(
        ...
        '230.3.2.1'
    );
    // ...
})();
```

Nothing stops the client from broadcasting to others in the group, or even members of another group:

```
dgram.createSocket('udp4')
.on('message', function(message, remote) {
    var msg = new Buffer("Calling original group!");
    // 230.1.2.3 is the multicast address
    socket.send(    msg, 0, msg.length, multicastPort, '230.1.2.3');
})
.bind(multicastPort, '230.3.2.1');
```

Any Node process that has an address on our network interface can now listen on a UDP multicast address for messages, providing a fast and elegant inter-process communication system.

Using Amazon Web Services in your application

As a few thousand users become a few million users, as databases scale to terabytes of data, the cost and complexity of maintaining an application begins to overwhelm teams with insufficient experience, funding, and/or time. When faced with rapid growth it is sometimes useful to delegate responsibilities for one or more aspects of your application to cloud-based service providers. **AWS (Amazon Web Services)** is just such a suite of cloud-computing services, offered by `amazon.com`.

You will need an AWS account in order to use these examples. All of the services we will explore are free or nearly free for low-volume development uses. To create an account on AWS, visit the following link:

`http://aws.amazon.com/`

Once you have created an account, you will be able to manage all of your services via the AWS console at:

`https://console.aws.amazon.com/console/home`

In this section we will learn how to use three popular AWS services:

- For storing documents and files we will connect with Amazon **S3 (Simple Storage Service)**
- Amazon's Key/Value database, DynamoDB
- To manage a large volume of e-mail, we will leverage Amazon's **SES (Simple Email Service)**

To access these services we will use the AWS SDK for Node, which can be found at the following link:

`https://github.com/aws/aws-sdk-js`

To install the module run the following command:

```
npm install aws-sdk
```

Full documentation for the `aws-sdk` module can be found at the following link:

`http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/frames.html`

Authenticating

Developers registered with AWS are assigned two identifiers:

- A public **Access Key ID** (a 20-character, alphanumeric sequence).
- A **Secret Access Key** (a 40-character sequence). It is very important to keep your Secret Key private.

Amazon also provides developers with the ability to identify the region with which to communicate, such as `"us-east-1"`. This allows developers to target the closest servers (regional endpoint) for their requests.

The regional endpoint and both authentication keys are necessary to make requests.

For a breakdown of regional endpoints visit the following link:

`http://docs.aws.amazon.com/general/latest/gr/rande.html`

As we will be using the same credentials in each of the following examples, let's create a single config.json file that is re-used:

```
{
    "accessKeyId"    : "your-key",
    "secretAccessKey"    : "your-secret",
    "region"        : "us-east-1",
    "apiVersions"    : {
          "s3"        : "2006-03-01",
          "ses"        : "2010-12-01",
          "dynamodb"    : "2012-08-10"
    }
}
```

We also configure the specific API versions we will be using for services. Should Amazon's services API change, this will ensure our code will continue to work.

An AWS session can now be initialized with just two lines of code. Assume that these two lines exist prior to any of the example code that follows:

```
var AWS = require('aws-sdk');
AWS.config.loadFromPath('./config.json');
```

Errors

When experimenting with these services it is likely that error codes will appear on occasion. Because of their complexity, and the nature of cloud computing, these services can sometimes emit surprising or unexpected errors. For example, because S3 can only promise eventual consistency in some regions and situations, attempting to read a key that has just been written to may not always succeed. We will be exploring the complete list of error codes for each service, and they can be found at the following locations:

- **S3**

  `http://docs.aws.amazon.com/AmazonS3/latest/API/ErrorResponses.html`

- **DynamoDB**

  `http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/Welcome.html`

- **SES**

  `http://docs.aws.amazon.com/ses/latest/DeveloperGuide/api-error-codes.html`

  and

  `http://docs.aws.amazon.com/ses/latest/DeveloperGuide/smtp-response-codes.html`

As it will be difficult in the beginning to predict where errors might arise, it is important to employ the `domain` module or other error-checking code as you proceed.

Additionally, a subtle but fundamental aspect of Amazon's security and consistency model is the strict synchronization of its web server time and time as understood by a server making requests. A discrepancy of 15 minutes is the maximum allowed. While this seems like a long time, in fact time drift is very common. When developing watch out for 403: Forbidden errors that resemble one of the following:

- `SignatureDoesNotMatch`: This error means that the signature has expired
- `RequestTimeTooSkewed`: The difference between the request time and the current time is too large

If such errors are encountered the internal time of the server making requests may have drifted. If so, that server's time will need to be synchronized. On Unix one can use the **NTP (Network Time Protocol)** to achieve synchrony. One solution is to use the following commands:

```
rdate 129.6.15.28
ntpdate 129.6.15.28
```

For more information on NTP and time synchronization visit the following link:

`http://www.pool.ntp.org/en/`

Let's start using AWS services, beginning with the distributed file service, S3.

Using S3 to store files

S3 can be used to store any file one expects to be able to store on a filesystem. Most commonly, it is used to store media files such as images and videos. S3 is an excellent document storage system as well, especially well-suited for storing small JSON objects or similar data objects.

Also, S3 objects are accessible via HTTP, which makes retrieval very natural, and REST methods such as PUT/DELETE/UPDATE are supported. S3 works very much like one would expect a typical file server to work, is spread across servers that span the globe, and offers storage capacity that is for all practical purposes limitless.

S3 uses the concept of a **bucket** as a sort of corollary to "hard drive". Each S3 account can contain 100 buckets (this is a hard limit), with no limits on the number of files contained in each bucket.

Working with buckets

Creating a bucket is easy:

```
var S3 = new AWS.S3();
S3.createBucket({
    Bucket: 'nodejs-book'
}, function(err, data) {
    if(err) { throw new Error("Unable to create bucket."); }
    console.log(data);
});
```

We will receive a data map containing the bucket `Location`, and a `RequestId`:

```
{    Location: '/nodejs-book',
RequestId: 'A0F2B68C72BE8A29' }
```

It is likely that many different operations will be made against a bucket. As a convenience the `aws-sdk` allows a bucket name to be automatically defined in the parameter list for all further operations:

```
var S3 = new AWS.S3({
    params: { Bucket: 'nodejs-book' }
});
S3.createBucket(function(err, data) { // ... });
```

Use `listBuckets` to fetch an array of existing buckets:

```
S3.listBuckets(function(err, data) {
    console.log(data.Buckets);
});
//    [ { Name: 'nodejs-book',
//        CreationDate: Mon Jul 15 2013 22:17:08 GMT-0500 (CDT) },
//        ...
//    ]
```

Bucket names are global to all S3 users. No single user of S3 can use a bucket name that another user has claimed. If I have a bucket named `foo` no other S3 user can ever use that bucket name. This is a gotcha that many miss.

Working with objects

Let's add a document to the `nodejs-book` bucket on S3:

```
var S3 = new AWS.S3({
    params: { Bucket: 'nodejs-book'    }
});
var body = JSON.stringify({ foo: "bar" });
var s3Obj = {
    Key: 'demos/putObject/first.json',
    Body: body,
    ServerSideEncryption: "AES256",
    ContentType: "application/json",
    ContentLength: body.length,
    ACL: "private"
}
```

```
S3.putObject(s3Obj, function(err, data) {
    if(err) { throw new Error("PUT error"); }
    console.log(data);
});
```

If the PUT is successful its callback will receive an object similar to the following:

```
{ ETag: '"9bb58f26192e4ba00f01e2e7b136bbd8"',
      ServerSideEncryption: 'AES256',
      RequestId: 'C7AAD7FF0A14F979' }
```

You are encouraged to consult the SDK documentation and experiment with all of the parameters `putObject` accepts. Here we focus on the only two required fields, and a few useful and common ones:

- Key: A name to uniquely identify your file within this bucket.
- Body: A Buffer, String, or Stream comprising the file body.
- ServerSideEncryption: Whether to encrypt the file within S3. The only current option is AES256 (which is a good one!).
- ContentType: Standard MIME type.
- ContentLength: A String indicating the destination IP for the datagram.
- ACL: Canned access permissions, such as `private` or `public-read-write`. Consult the S3 documentation.

It is a good idea to have the object **Key** resemble a filesystem path, helping with sorting and retrieval later on. In fact, Amazon's S3 console reflects this pattern in its UI:

Let's stream an image up to S3:

```
var fs = require('fs');
fs.stat("./testimage.jpg", function(err, stat) {
    var s3Obj = {
        Key: 'demos/putObject/testimage.jpg',
        Body: fs.createReadStream("./testimage.jpg"),
        ContentLength: stat.size,
        ContentType: "image/jpeg",
        ACL: "public-read"
    }
    S3.putObject(s3Obj, function(err, data) { // ... });
})
```

As we gave this image `public-read` permissions, it will be accessible at the following URL:

```
https://s3.amazonaws.com/nodejs-book/demos/putObject/testimage.jpg
```

Fetching an object from S3 and streaming it onto a local filesystem is even easier:

```
var outFile = fs.createWriteStream('./fetchedfile.jpg');
S3.getObject({
    Key    : 'demos/putObject/testimage.jpg'
}).createReadStream().pipe(outFile);
```

Alternatively, we can catch data events on the HTTP chunked transfer:

```
S3.getObject({
    Key    : 'demos/putObject/testfile.jpg'
})
.on('httpData', function(chunk) {     outFile.write(chunk); })
.on('httpDone', function() { outFile.end(); })
.send();
```

To delete an object:

```
S3.deleteObject({
    Bucket : 'nodejs-book',
    Key     : 'demos/putObject/optimism.jpg'
}, function(err, data) { //     ... });
```

To delete multiple objects, pass an Array (to a maximum of 1000 objects):

```
S3.deleteObjects({
    Bucket    : 'nodejs-book',
    Delete    : {
        Objects    : [{
            Key    : 'demos/putObject/first.json'
        }, {
            Key    : 'demos/putObject/testimage2.jpg'
        }]
    }
}, function(err, data) { //    ... });
```

Using AWS with a Node server

Putting together what we know about Node servers, streaming file data through pipes, and HTTP, it should be clear how to mount S3 as a filesystem in just a few lines of code:

```
http.createServer(function(request, response) {
    var requestedFile = request.url.substring(1);
    S3.headObject({
        Key : requestedFile
    }, function(err, data) {
        //    404, etc.
        if(err) {
            response.writeHead(err.statusCode);
            return response.end(err.name);
        }
        response.writeHead(200, {
            "Last-Modified"     : data.LastModified,
            "Content-Length"     : data.ContentLength,
            "Content-Type"     : data.ContentType,
            "ETag"         : data.ETag
        });
        S3.getObject({
            Key    : requestedFile
        }).createReadStream().pipe(response);
    });
}).listen(8080);
```

A standard Node HTTP server receives a request URL. We first attempt a HEAD operation using the `aws-sdk` method `headObject`, accomplishing two things:

- We'll determine if the file is available
- We will have the header information necessary to build a response

After handling any non-200 status code errors we only need to set our response headers and stream the file back to the requester, as previously demonstrated.

Such a system can also operate as a **fail-safe**, in both directions; should S3, or the file, be unavailable, we might bind to another file system, streaming from there. Conversely, if our preferred local filesystem fails, we might fall through to our backup S3 filesystem.

See the `amazon/s3-redirect.js` file in the code bundle available at the Packt website for an example of using 302 redirects to similarly mount an AWS filesystem.

S3 is a powerful data storage system with even more advanced features than those we've covered, such as object versioning, download payment management, and setting up objects as torrent files. With its support for streams, the `aws-sdk` module makes it easy for Node developers to work with S3 as if it was a local filesystem.

Getting and setting data with DynamoDB

DynamoDB (**DDB**) is a NoSQL database providing very high throughput and predictability that can be easily scaled. DDB is designed for **data-intensive** applications, performing massive map/reduce and other analytical queries with low latency and reliably. That being said, it is also an excellent database solution for general web applications.

The whitepaper announcing DynamoDB was highly influential, sparking a real interest in NoSQL databases, and inspiring many, including Apache **Cassandra**. The paper deals with advanced concepts, but rewards careful study:

`http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf`

A Dynamo database is a collection of tables, which is a collection of items, which are a collection of attributes. Each item in a table (or row, if you prefer) must have a primary key, functioning as an index for the table. Each item can have any number of attributes (up to a limit of 65 KB) in addition to the primary key.

This is an item, with five attributes, one attribute serving as the primary key (`Id`):

```
{    Id = 123
       Date = "1375314738466"
       UserId = "DD9DDD8892"
       Cart = [ "song1", "song2" ]
       Action = "buy" }
```

Let's create a table with both a primary and a secondary key:

```
var AWS = require('aws-sdk');
AWS.config.loadFromPath('../config.json');
var db = new AWS.DynamoDB();
db.createTable({
    TableName: 'purchases',
    AttributeDefinitions : [{
        AttributeName : "Id", AttributeType : "N"
    }, {
        AttributeName : "Date", AttributeType : "N"
    }],
```

```
    KeySchema: [{
        AttributeName: 'Id',  KeyType: 'HASH'
    }, {
        AttributeName: 'Date',  KeyType: 'RANGE'
    }],
    ProvisionedThroughput: {
        ReadCapacityUnits: 2,
        WriteCapacityUnits: 2
    }
}, function(err, data) { console.log(util.inspect(data)); });
```

The callback will receive an object similar to this:

```
{     TableDescription: {
        AttributeDefinitions: [ // identical to what was sent],
        CreationDateTime: 1375315748.029,
        ItemCount: 0,
        KeySchema: [ //    identical to what was sent ],
        ProvisionedThroughput: {
            NumberOfDecreasesToday: 0,
            ReadCapacityUnits: 2,
            WriteCapacityUnits: 2
        },
        TableName: 'purchases',
        TableSizeBytes: 0,
        TableStatus: 'CREATING'
    }
}
```

Table creation/deletion is not immediate—you are essentially queueing up the creation of a table (note `TableStatus`). At some point in the (near) future the table will exist. As DDB table definitions cannot be changed without deleting the table and rebuilding it, in practice, this delay is not something that should impact your application—build once, then work with items.

DDB tables must be given a schema indicating the item attributes that will function as keys, defined by `KeySchema`. Each attribute in `KeySchema` can be either a `RANGE` or a `HASH`. There must be one such index; there can be at most two. Each added item must contain any defined keys, with as many additional attributes as desired.

Each item in `KeySchema` must be matched in count by the items in `AttributeDefinitions`. In `AttributeDefinitions` each attribute can be either a number (`"N"`) or a string (`"S"`). When adding or modifying attributes it is always necessary to identify attributes by its type as well as by the name.

To add an item:

```
db.putItem({
    TableName : "purchases",
    Item : {
        Id : {"N": "123"},
        Date : {"N": "1375314738466"},
        UserId : {"S" : "DD9DDD8892"},
        Cart : {"SS" : [ "song1", "song2" ]},
        Action : {"S" : "buy"}
    }
}, function() { // ... });
```

In addition to our primary and (optional) secondary keys we want to add other attributes to our item. Each must be given one of the following types:

- S: A String
- N: A Number
- B: A Base64-encoded string
- SS: An Array of Strings (String set)
- NS: An Array of Numbers (Number set)
- BS: An Array of Base64-encoded strings (Base64 set)

All items will need to have this same number of columns—again, dynamic schemas are *not* a feature of DDB.

Assume that we've created a table that looks like the following table:

Id	Date	Action	Cart	UserId
123	1375314738466	buy	{ "song1", "song2" }	DD9DDD8892
124	1375314738467	buy	{ "song2", "song4" }	DD9EDD8892
125	1375314738468	buy	{ "song12", "song6" }	DD9EDD8890

Now let's perform some search operations.

Searching the database

There are two types of search operations available: **query** and **scan**. A scan on a table with a single primary key will, without exception, search every item in a table, returning those matching your search criteria. This can be very slow on anything but small databases. A query is a direct key lookup. We'll look at queries first. Note that in this example we will assume that this table has only one primary key.

To fetch the `Action` and `Cart` attributes for item `124`, we use the following code:

```
db.getItem({
    TableName : "purchases",
    Key : {
        Id : { "N"    : "124" }
    },
    AttributesToGet : ["Action", "Cart"]
}, function(err, res) {
    console.log(util.inspect(res, { depth: 10 }));
});
```

Which will return:

```
{     Item: {
        Action: { S: 'buy' },
        Cart: { SS: [ 'song2', 'song4' ] }
    }
}
```

To select all attributes, simply omit the `AttributesToGet` definition.

A scan is more expensive, but allows more involved searches. The usefulness of secondary keys is particularly pronounced when doing scans, allowing us to avoid the overhead of scanning the entire table. In our first example of scan we will work as if there is only a primary key. Then we will show how to filter the scan using the secondary key.

To get all the records whose `Cart` attribute contains `song2`, use the following code:

```
db.scan({
    TableName : "purchases",
    ScanFilter : {
          "Cart": {
              "AttributeValueList" : [{
                  "S":"song2"
              }],
            "ComparisonOperator" : "CONTAINS"
        },
    }
```

```
}, function(err, res) {
    console.log(util.inspect(res, {
        depth: 10
    }));
});
```

This will return all attribute values for items with `Id` 123 and 124.

Let's now use our secondary key to filter this further:

```
db.scan({
    TableName : "purchases",
    ScanFilter : {
        "Date": {
            "AttributeValueList" : [{
                "N" : "1375314738467"
            }],
            "ComparisonOperator" : "EQ"
        },
          "Cart": {
              "AttributeValueList" : [{
                  "S" : "song2"
              }],
            "ComparisonOperator" : "CONTAINS"
        },
    }
}, function(err, res) {
    console.log(util.inspect(res, {depth: 10}));
});
```

This new filter limits results to item 124.

Sending mail via SES

Amazon describes the problems SES is designed to solve in this way:

> *Building large-scale email solutions to send marketing and transactional messages is often a complex and costly challenge for businesses. To optimize the percentage of emails that are successfully delivered, businesses must deal with hassles such as email server management, network configuration, and meeting rigorous Internet Service Provider (ISP) standards for email content.*

Apart from the typical network scaling problems inherent in growing any system, providing e-mail services is made particularly difficult due to the prevalence of spam. It is very hard to send a large number of unsolicited e-mails without ending up blacklisted, even when the recipients are amenable to receiving them. Spam control systems are automated; your service must be listed in the "whitelists", which is used by various e-mail providers and spam trackers in order to avoid having a low percentage of your e-mails end up somewhere other than your customer's inbox. A mail service must have a good reputation with the right people or it becomes nearly useless.

Amazon's SES service has the necessary reputation, providing application developers with cloud-based e-mail service which is reliable and able to handle a nearly infinite volume of e-mail. In this section we will learn how SES can be used by a Node application as a reliable mail delivery service.

Ensure that you have SES access by visiting your developer console. When you first sign up with SES you will be given *Sandbox* access. When in this mode you are limited to using only Amazon's mailbox simulator, or sending e-mail to address you have verified (such as one's own). You may request production access, but for our purposes you will only need to verify an e-mail address to test with.

Because the cost of using a service such as SES will increase as your mail volume increases, you might want to periodically check your quotas:

```
var ses = new AWS.SES();
ses.getSendQuota(function(err, data) {
    console.log(err, data);
});
```

To send a message:

```
ses.sendEmail({
    Source : "spasquali@gmail.com",
    Destination    : {
        ToAddresses : [ "spasquali@gmail.com" ]
    },
    Message    : {
        Subject: { Data : "NodeJS and AWS SES" },
        Body : {
            Text : { Data : "It worked!" }
        }
    }
}, function(err, resp) {
    console.log(resp);
});
```

The callback will receive something like the following output:

RequestId: '623144c0-fa5b-11e2-8e49-f73ce5ee2612'

MessageId: '0000014037f1a167-587a626e-ca1f-4440-a4b0-81756301bc28-000000'

Multiple recipients, HTML body contents, and all the other features one would expect from a mail service are available.

Authenticating with Facebook Connect

Building a scalable authentication system is hard. Though it may seem easy—create a table or a list of users and their credentials, change that list using a simple web form, and you're done!

If your goal is to create a scalable system that can grow as your user base grows then you will need to do much more. It will be necessary to create e-mail systems and scalable databases, session management, and the many other subsystems essential to satisfying the user's expectation of a low-latency path from sign-up to sign-in to the desired content.

Let's put together what we've learned with AWS with Connect, Facebook's authentication service. You can see the full code for this example in the `/facebook` folder of your code bundle available at the Packt website.

To begin, visit `developers.facebook.com`, set up a developers account, and then create an application. Remember to properly set your application path as shown in the following screenshot:

Once registered, find and copy your two application keys. You will need these later.

To create a Node server that allows users to login via Connect, we will use the excellent `passport` npm module, which will make it easy to connect to many cloud services, including Facebook. Additionally, we will use S3 to store a simple user object, allowing us to map Connect login IDs to our internal application records. We'll build this all using Express, which works well with `passport`.

Initialize S3 as per our previous examples, and create a bucket to store user records in. `passport` works with the metaphor of a "strategy". Once we've established our credentials using our Facebook login keys, we provide a callback to catch the results of a user's attempt to log in—Facebook will redirect to this URL after a user has attempted authentication.

If the login is successful, map a record of this user to our S3 store, setting the S3 object key to the user's Facebook ID. We now have a strategy in place to accept logins using Connect:

```
var FacebookStrategy = require('passport-facebook').Strategy;
//    For the demo we use a simple session strategy.
//    You will likely handle sessions using a DB.
passport.serializeUser(function(user, done) {
    done(null, user);
});
passport.deserializeUser(function(obj, done) {
    done(null, obj);
});
passport.use(new FacebookStrategy({
    clientID    : 'your-id',
    clientSecret    : 'your-secret',
    callbackURL    : "http://www.pathjs.com/auth/facebook/callback"
},
function(accessToken, refreshToken, profile, done) {
    profile.local = {
        lastLogin : new Date().getTime(),
        sessionId : parseInt(Math.random() * 100000)
    }
    var s3Obj = {
        Key: profile.id,
        Body: JSON.stringify(profile),
        ServerSideEncryption    : "AES256",
        ContentType: "application/json",
        ACL: "private"
    }
    S3.putObject(s3Obj, function(err, data) {
        if(err) { return done(err); }
        return done(err, profile);
    });
}));
```

When a user visits our site, we can now present them with a Facebook login.

All that is left to do is create an **Express** server with the necessary paths to authenticate and log out, including a path to handle the Facebook callback:

```
var app = express();
app.configure(function() {
    app.set('views', __dirname + '/views');
    app.set('view engine', 'jade');
    // more setup
});
app.get('/', function(req, res){
    res.render('index', {
        user: req.user
    });
});
app.get('/auth/facebook', passport.authenticate('facebook'));
app.get(
    '/auth/facebook/callback',
    passport.authenticate('facebook', {
        failureRedirect: '/'
    }),
    function(req, res) {
        res.redirect('/');
      }
);
app.get('/logout', function(req, res){
    req.logout();
    res.redirect('/');
});
app.listen(8080);
```

One final step is to use the **Jade** templating engine to create the login/logout pages (see the **Express** initialization). The key bit is passing responsibility for rendering our application's `index` page, located at the server root, to a Jade template:

```
app.get('/', function(req, res){
    res.render('index', {
        user: req.user
    });
});
```

This template can be found in the `index.jade` file located in `facebook/views/` within your code bundle.

That's it! Your Node application, already fast, has now delegated responsibility for authentication and user record storage to experts in scaling (and securing) such systems. Your application can now handle several thousands of simultaneous connections *and* scale nearly indefinitely by leveraging two of the largest and most stable application infrastructures in the world—one for free, and the other at a very low cost.

Summary

Big data applications have placed significant responsibility on developers of network applications to prepare for scale. Node has offered help in creating a network-friendly application development environment that can easily connect to other devices on a network, such as cloud services and, in particular, other Node servers.

In this chapter we learned some good strategies for scaling Node servers, from analyzing CPU usage to communicating across processes. With our new knowledge of message queues and UDP we can build networks of Node servers scaling horizontally, letting us handle more and more traffic by simply replicating existing nodes. Having investigated load balancing and proxying with Node, we can confidently add capacity to our applications. When matched with the cloud services provided by AWS and Facebook, we are able to scale our data stores and user account management with ease and at low cost.

As our applications grow we will need to maintain continuous awareness of how each part, as well as the whole, is behaving. Before a new component is added, that integration will need to be tested. In the next chapter we will learn strategies for testing your systems, both in the large and in the small.

9
Testing your Application

When the terrain disagrees with the map, trust the terrain.

– Swiss Army Manual

Because a community fully committed to code sharing is building Node, and interoperability between modules is so important, it should come as no surprise that code testing tools and frameworks entered Node's ecosystem right after inception. Indeed, the normally parsimonious core Node team added the `assert` module early on, suggesting a recognition that testing is a fundamental part of the development process.

Testing software is a complicated and still mostly ill-defined activity. It is also essential to all phases of a non-trivial development project. The set of expectations accompanying the adjective "enterprise" when associated with "software" is a large one, comprising at least security, stability, consistency, predictability, and scale. Software is expected to anticipate and nimbly integrate multifaceted and unpredictable changes in data volume and shape, user expectations, business goals, staffing, even government regulations, and legal climate.

Testing is not solely a bug-detecting and defect-fixing process. Test-driven development, for example, insists on having tests precede the existence of any code! Testing, generally, is the process of making comparisons between existing behavior and desired behavior in software, where new information is continuously fed back into the process. In this sense, testing involves modeling expectations and verifying that individual functions, composed units, and implementation paths satisfy the expectations of designers, programmers, product owners, customers, and entire organizations.

Achieving this goal requires that the pieces of a system are well tested, such that their inclusion or exclusion will have a known and consistent impact on both the system itself and on expectations for the system, such as sales, or implementation velocity, or ability to process secure data. Testing is therefore also about managing risk. In this way anomalies can be quantified and qualified, while bumps in the terrain can now usefully inform our current understanding of the map such that the number of missteps (or defects) decline and our confidence rises. Testing helps us measure when we are done.

There are many schools of thought on when to test, what to test, who should test, and how to test. In this chapter we will focus on some known and useful patterns for testing Node applications, investigating native Node tools for code integrity testing, general testing with the Mocha framework, and headless browser testing; the last allowing for the testing of browser-based JavaScript from within a Node environment.

As you move through this chapter, it might be useful to keep in mind that integrating the philosophy of testing into a project is very difficult to do well. Typically, implementation becomes more difficult as the size of an untested codebase increases, and/or as the number of contributors grow. Some might say that a comprehensive testing strategy should be implemented as early as possible—something to consider as you embark on your next Node project.

Why testing is important

A good testing strategy builds confidence through the accumulation of proof and increasing clarity. Within a company this might mean that some criteria for the execution of a business strategy have been satisfied, allowing for the release of a new product. The developers within a project team gain the pleasure of an automated judge that confirms or denies whether changes committed to a codebase are sound. With a good testing framework refactoring loses its danger: the "if you break it you own it" caveat that once placed negative pressure on developers with new ideas is no longer as ominous. Given a good version control system and test/release process, any breaking change can be rolled back without negative impact, freeing curiosity and experimentation.

Three common types of tests are: unit tests, functional tests, and integration tests. While our goal in this chapter is not to put forward a general theory about how to test applications, it will be useful to briefly summarize what unit, functional, and integration tests are, which members of a team are most interested in each, and how we might go about breaking up a codebase into testable units.

Unit tests

Unit tests concern themselves with units of system behavior. Each unit being tested should encapsulate a very small set of code paths, without entanglements. When a unit test fails this should, ideally, indicate that an isolated part of the overall functionality is broken. If a program has a well described set of unit tests, the purpose and expected behavior of an entire program should be easy to comprehend, entire program should be easy to comprehend. A unit test applies a limited perspective to small parts of a system, unconcerned with how those parts may be wrapped up into larger functional blocks.

An example unit test might be described in this way: when the value `123` is passed to a method `validate_phone_number()`, the test should return false. There is no confusion about what this unit does, and a programmer can use it with confidence.

Unit tests are normally written and read by programmers. Class methods are good candidates for unit tests, as are other service endpoints whose input signatures are stable and well understood, with expected outputs that can be accurately validated. Generally it is assumed that unit tests run quickly. If a unit test is taking a long time to execute it is likely the code under test is much more complex than it should be.

Unit tests are not concerned with how a function or method under test will receive its inputs, or how it will be used in general. A test for an `add` method shouldn't be concerned with whether the method will be used in a calculator or somewhere else, it should simply test if the two integer inputs (3,4) will cause the unit to emit a correct result (7). A unit test is not interested in where it fits in the dependency tree. For this reason unit tests will often "mock" or "stub" data sources, such as passing two random integers to an `add` method. As long as the inputs are typical they need not be actual. Additionally, good unit tests are reliable: unencumbered by external dependencies they should remain valid regardless of how the system around them changes.

Unit tests only confirm that a single entity works in isolation. Testing whether units can work well when combined is the purpose of functional testing.

Functional tests

Where unit tests concern themselves with specific behaviors, functional tests are designed to validate pieces of functionality. The ambiguity of the root word "function", especially for programmers, can lead to confusion, where "unit tests" are called "functional tests", and vice versa. A functional test combines many units into a body of functionality, such as, "when a user enters a username and password and clicks on send, that user will be logged into the system". We can easily see that this functional group would be comprised of many unit tests, one for validating a username, one for handling a button click, and so on.

Functional tests are normally the concern of those responsible for some specific domain within an application. While programmers and developers will remain the ones to implement these tests, product managers or similar stakeholders will normally devise them. These tests for the most part check whether larger product specifications are being satisfied, rather than for technical correctness.

The example unit test for `validate_phone_number` given earlier might form part of a functional test with this description: when a user enters the wrong phone number, open a help dialog that describes the right format in that user's country. That an application bothers to help users that make mistakes with phone numbers is an abstract effort very different from simply validating a technical entity like a phone number. Functional tests might be thought of as abstract models of how well some collection of units work together to satisfy a product need.

Because functional tests are made against combinations of many units, it is expected that, unlike an isolated unit test, executing them will involve mixing concerns from any number of external objects or systems. In the preceding login example we see how a relatively simple functional test can cut across database, UI, security, and other application layers. Because it is compositionally more complex, a functional test is expected to take a little more time to run than a unit test. Functional tests are expected to change less often than unit tests, such that changes in functionality often represent major releases, as opposed to the minor changes unit test modifications usually indicate.

Note that, like unit tests, functional tests are themselves isolated from concerns about how the functional group under test, as a whole, relates to the rest of an application. For this reason mock data may be used as a context for running functional tests, as the functional group itself is not concerned with its effect on general application state, which is the domain of integration tests.

Integration tests

Integration tests ensure that the entire system is correctly wired together, such that a user would feel the application is working correctly. In this way integration tests typically validate the expected functionality of an entire application, or one of a small set of significant product functionality.

The most important difference between integration and the other types of tests under discussion is that integration tests are to be executed within a realistic environment, on real databases with actual domain data, on servers, and other systems mirroring the target production environment. In this way, integration tests can easily break formerly passing unit and functional tests.

For example, a unit test for `validate_phone_number` may have given a pass to an input like `555-123-4567`, but during an integration test it will fail to pass some real (and valid) system data like `555.123.4567`. Similarly, a functional test may successfully test the ability of an ideal system to open a help dialog, but when integrated with a new browser or other runtime it is found that the expected functionality is not achieved. An application that runs well against a single local filesystem may fail when run against a distributed filesystem.

Because of this added complexity, system architects—the team members that are able to apply a higher-level perspective on what system correctness entails—normally design integration tests. These tests find errors in the wiring that isolated tests are not able to recognize. Not surprisingly, integration tests can often take a long time to run, typically designed to not only run simple scenarios but to imitate expected high-load, realistic environments.

A powerful new tool for doing deep testing of Node application performance is DTrace. Joyent's hosting cloud provides access to this tool (`http://www.joyent.com/blog/bruning-questions-debugging-node-apps-with-dtrace`), and support is growing. A nice overview of DTrace can be found here:

`http://mcavage.github.com/presentations/dtrace_conf_2012-04-03/#agenda`

Native Node testing and debugging tools

A preference for tested code has formed part of the Node community's ethos since its inception, reflected in the fact that most popular Node modules, even simple ones, are distributed with test suites. While browser-side development with JavaScript suffered for many years without usable testing tools, the relatively young Node distribution contains many. Perhaps because of this, many mature and easy to use third-party testing frameworks have been developed for Node. This leaves a developer no excuse for writing untested code! Let's look into some of the provided tools for debugging and testing Node programs.

Writing to the console

Console output is the most basic testing and debugging tool, providing a quick way to see what is happening at some point in a script. The globally accessible `console.log` is commonly used when debugging.

Node has enriched this standard output mechanism with more useful methods, such as `console.error(String, String…)`, which prints arguments to `stderr` rather than `stdout`, and `console.dir(Object)`, which runs `util.inspect` (see the following) on the provided object and writes results to `stdout`.

The following pattern is commonly seen when a developer wants to track how long a piece of code takes to execute:

```
var start = new Date().getTime();
for(x=0; x < 1000; x++) { ... }
console.log(new Date().getTime() - start);
// A time, in milliseconds
```

The `console.time` and `console.timeEnd` methods standardize this pattern:

```
console.time('Add 1000000 records');
var rec = [];
for (var i = 0; i < 1000000; i++) {
  rec.push(1);
}
console.timeEnd('Add 1000000 records');
//  > Add 1000000 records: 59ms
```

We will see other special console methods when discussing the `assert` module and performing stack traces later in this chapter.

More output tools are accessible via the `util` module:

- `util.debug(String)`: Blocks the running process and outputs the string to `stderr`:

  ```
  > require('util').debug("Error!")
  DEBUG: Error!
  ```

- `util.error(String, [String…])`: Same as debug, but can send multiple strings to `stderr`:

  ```
  > require('util').error("Error1","Error2","Error3")
  Error1
  Error2
  Error3
  ```

- `util.puts(String, [String…])`: Outputs multiple strings to `stdout`, adding a newline after each argument:

  ```
  > require('util').puts("A","B","C")
  A
  B
  C
  ```

- `util.print(String, [String...])`: Outputs multiple strings to `stdout`, without adding a newline after each argument:

```
> require('util').print(1,2,3)
123
```

- `util.log(String)`: Outputs a timestamped message to `stdout`:

```
> require('util').log('Message with a timestamp')
20 Aug 18:32:08 - Message with a timestamp
```

Formatting console output

The preceding methods are all very useful when logging simple strings. More often, useful logging data may need to be formatted, either by composing several values into a single string, or by neatly displaying a complex data object. The `util.format` and `util.inspect` methods can be used to handle these cases.

The util.format(format, [arg, arg…]) method

This method allows a formatting string to be composed out of placeholders, each of which captures and displays the additional values passed. For example:

```
> util.format('%s:%s', 'foo','bar')
'foo:bar'
```

Here, we see that the two placeholders (prefixed by `%`) are replaced in order by the passed arguments. Placeholders expect one of the following three types of values:

- `%s`: A string
- `%d`: A number, either an integer or a float
- `%j`: A JSON object

If a greater number of arguments than placeholders is sent, the extra arguments are converted to strings via `util.inspect()`, and concatenated to the end of the output, separated by spaces:

```
> util.format('%s:%s', 'foo', 'bar', 'baz');
'foo:bar baz'
```

If no formatting string is sent, the arguments are simply converted to strings and concatenated, separated by a space.

The util.inspect(object, [options]) method

Use this method when a string representation of an object is desired. Through the setting of various options the look of the output can be controlled:

- `showHidden`: Defaults to false. If true, the object's non-enumerable properties will be shown.
- `depth`: An object definition, such as a JSON object, can be deeply nested. By default `util.inspect` only traverses two levels into the object. Use this option to increase (or decrease) that depth.
- `colors`: Allows the colorization of the output (see the following code snippet).
- `customInspect`: If the object being processed has an `inspect` method defined, the output of that method will be used instead of Node's default stringification method (see the following code snippet). Defaults to true.

Setting a custom inspector:

```
var util = require('util');
var obj = function() {
  this.foo = "bar";
};
obj.prototype.inspect = function() {
  return "CUSTOM INSPECTOR";
}
console.log(util.inspect(new obj))
//  CUSTOM INSPECTOR
console.log(util.inspect(new obj, { customInspect: false }))
//  { foo: 'bar' }
```

This can be very useful when logging complex objects, or objects whose values are so large as to make console output unreadable.

The `color: true` option of `util.inspect` allows for colorization of object output, which can be useful for complex objects. Colors are assigned to data types. The default assignments are set in `util.inspect.styles`:

```
{ special: 'cyan',
  number: 'yellow',
  boolean: 'yellow',
  undefined: 'grey',
  null: 'bold',
  string: 'green',
  date: 'magenta',
  regexp: 'red' }
```

These default color assignments may be swapped out with one of the supported ANSI color codes stored in the `util.inspect.colors` object: bold, italic, underline, inverse, white, grey, black, blue, cyan, green, magenta, red, and yellow. For example, to have the number values of objects displayed in `green` rather than the default of `yellow` use the following code:

```
util.inspect.styles.number = "green";
console.log(util.inspect([1,2,4,5,6], {colors: true}));
// [1,2,3,4,5,6] Numbers are in green
```

The Node debugger

Most developers have used an IDE for development. A key feature of all good development environments is access to a debugger, which allows breakpoints to be set in a program in places where state or other aspects of the runtime need to be checked.

V8 is distributed with a powerful debugger (commonly seen powering the Google Chrome browser's developer tools panel), and this is accessible to Node. It is invoked using the debug directive:

```
> node debug somescript.js
```

Simple step-through and inspection debugging can now be achieved within a node program. Consider the following program:

```
myVar = 123;
setTimeout(function () {
  debugger;
  console.log("world");
}, 1000);
console.log("hello");
```

Note the `debugger` directive. Executing this program without using the `debugger` directive will result in `hello` being displayed, followed by `world`, one second later. When using the directive, one would see this:

```
> node debug somescript.js
< debugger listening on port 5858
connecting... ok
break in debug-sample.js:1
  1 myVar = 123;
  2 setTimeout(function () {
  3 debugger;
debug>
```

Once a breakpoint is hit, we are presented with a CLI to the debugger itself, from within which we can execute some standard debugging and other commands:

- `cont` or `c`: Continue execution from last breakpoint, until next breakpoint
- `step` or `s`: Step in, that is, keep running until a new source line (or breakpoint) is hit, then return control to debugger
- `next` or `n`: Same as `step`, but function calls made on the new source line are executed without stopping
- `out` or `o`: Step out, that is, execute the remainder of the current function and back out to the parent function
- `backtrace` or `bt`: Trace the steps to the current execution frame, similar to:

  ```
  …
  #3 Module._compile module.js:456:26
  #4 Module._extensions..js module.js:474:10
  #5 Module.load module.js:356:32
  … etc.
  ```

- `setBreakpoint()` or `sb()`: Set a breakpoint on the current line
- `setBreakpoint(Integer)` or `sb(Integer)`: Set a breakpoint on the specified line
- `clearBreakpoint()` or `cb()`: Clear breakpoint on current line
- `clearBreakpoint(Integer)` or `cb(Integer)`: Clear a breakpoint on the specified line
- `run`: If the debugger's script has terminated, this will start it again
- `restart`: Terminates and re-starts the script
- `pause` or `p`: Pause running code
- `kill`: Kill the running script
- `quit`: Exit the debugger
- `version`: Display V8 version
- `scripts`: Lists all loaded scripts

To repeat the last debugger command simply hit *Enter* on your keyboard.

Returning to the script we are debugging, entering `cont` into the debugger would result in the following output:

```
debug> cont
< hello // ... a pause of 1000 ms will now occur, then...
break in debug-sample.js:3
  1 myVar = 123;
```

```
  2 setTimeout(function () {
  3 debugger;
  4 console.log("world");
  5 }, 1000);
debug>
```

Notice how `hello` was not printed when we started the debugger, even though one would expect the command `console.log('hello')` to execute prior to the breakpoint being reached in the `setTimeout` callback. The debugger is not executing at run time: it is evaluating at compile time as well as at run time, giving you deep visibility into how the bytecode for your program is being assembled and eventually executed.

It is normally useful at a breakpoint to do some inspection, such as the value of variables. There is an additional command available to the debugger: `repl`, that enables this. Currently, our debugger is stopped after having successfully parsed the script and executed the first function pushed into the event loop, `console.log('hello')`. What if we wanted to check the value of `myVar`? Use `repl`:

```
debug> repl
Press Ctrl + C to leave debug repl
> myVar
123
```

Play around with REPL here, experimenting with how it might be used.

At this point our program has a single remaining instruction to execute, printing `world`. An immediate `cont` command will execute this last command, the event loop will have nothing further to do, and our script will terminate:

```
debug> cont
< world
program terminated
debug>
```

As an experiment, run the script again, using `next` instead of `cont`, just before the execution of this final context. Keep hitting *Enter* and try to follow the code that is being executed. You will see that after `world` is printed the `timers.js` script will be introduced into this execution context, as Node cleans up after firing a timeout. Run the `scripts` command in the debugger at this point. You will see something like:

```
debug> next
break in timers.js:125
 123
 124   debug(msecs + ' list empty');
 125   assert(L.isEmpty(list));
 126   list.close();
 127   delete lists[msecs];
debug> scripts
* 37: timers.js
  46: debug-sample.js
debug>
```

It will be useful to experiment with various methods, learning both what is happening when Node executes scripts at a deep level, as well as helping with your debugging needs.

It may be useful to read the following document, describing how the Google Chrome debugger interface is used at `https://developers.google.com/chrome-developer-tools/docs/javascript-debugging#breakpoints`.

Miroslav Bajtos' node-inspector module is something to consider for debugging, allowing a developer to remotely debug a Node application from the Chrome browser:

`https://github.com/node-inspector/node-inspector`

The assert module

Node's `assert` module is used for simple unit testing. In many cases it suffices as a basic scaffolding for tests, or used as the assertion library for testing frameworks (like Mocha, as we'll see later). Usage is straightforward: we want to assert the truth of something, and throw an error if our assertion is not true. For example:

```
> require('assert').equal(1,2,"Not equal!")
AssertionError: Not equal!
    at repl:1:20
  ...
```

If the assertion were true (both values are equal) nothing would be returned:

```
> require('assert').equal(1,1,"Not equal!")
undefined
```

Following the UNIX Rule of Silence (when a program has nothing surprising, interesting or useful to say, it should say nothing), assertions only return a value when the assertion fails. The value returned can be customized using an optional message argument, as seen in the preceding section.

The `assert` module API is composed of a set of comparison operations with identical call signatures: the actual value, the expected value, and an optional message to display when comparison fails. Alternate methods functioning as shortcuts or handlers for special cases are also provided.

A distinction must be made between identity comparison (`===`) and equality comparison (`==`) the former often referred to as strict equality comparison (as is the case in the `assert` API). Because JavaScript employs dynamic typing, when two values of different types are compared using the equality operator `==`, an attempt is made to coerce (or cast) one value into the other, a sort of common denominator operation. For example:

```
1 == "1" // true
false == "0" // true
false == null // false
```

As you might expect, these sorts of comparisons can lead to surprising results. Notice the more predictable results when identity comparison is used:

```
1 === "1" // false
false === "0" // false
false === null // false
```

The thing to remember is that the `===` operator does not perform type coercion prior to the comparison, while the equality operator compares after type coercion.

Additionally, because objects may contain the same values but not be derived from the same constructor, the identity of two objects with the same values is distinct for objects, identity requires that both operands refer to the same object:

```
var a = function(){};
var b = new a;
var c = new a;
var d = b;
console.log(a == function(){}) // false
console.log(b == c) // false
console.log(b == d) // true
console.log(b.constructor === c.constructor); // true
```

Finally, the concept of deep equality is used for object comparisons where identity need not be exact. Two objects are deeply equal if they both possess the same number of owned properties, the same prototype, the same set (though not necessarily the same order) of keys, and equivalent (not identical) values for each of their properties:

```
var a = [1,2,3];
var b = [1,2,3];
assert.deepEqual(a, b);// passes
assert.strictEqual(a, b);// throws AssertionError: [1,2,3] ===
  [1,2,3]
```

It is useful to test your assumptions about how values are understood in comparison to each other by designing some assertion tests. The results may surprise you. Node's assert module provides the following comparison methods:.

- `assert.equal(actual, expected, [message])`: Test coerced equality with `==`.
- `assert.notEqual(actual, expected, [message])`: Test coerced equality with `!=`.
- `assert.deepEqual(actual, expected, [message])`: Test for deep equality.
- `assert.notDeepEqual(actual, expected, [message])`: Test for deep inequality.
- `assert.strictEqual(actual, expected, [message])`: Test identity equivalence. Identical to using the `===` operator.
- `assert.notStrictEqual(actual, expected, [message])`: Test for identity mismatch. Identical to using the`!==` operator.
- `assert(value, [message])`: Throws error if sent value is not truthy.
- `assert.ok(value, [message])`: Identical to `assert(value)`.
- `assert.ifError(value)`: Throws error if value is truthy.

- `assert.throws(block, [error], [message])`: Test if the supplied code block throws an error. The optional error value can be an error constructor, regular expression, or a validation function returning a Boolean value.
- `assert.doesNotThrow(block, [error], [message])`: Test if the supplied code block does not throw an error.
- `assert.fail(actual, expected, message, operator)`: Throws an exception. This is most useful when the exception is trapped by a try/catch block:

```
try {
  assert.fail(1,2,"Bad!","NOT EQ")
} catch(e) {
  console.log(e)
}
```

The preceding code produces the following output:

```
{ message: 'Bad!',
  actual: 1,
  expected: 2,
  operator: 'NOT EQ',
  name: 'AssertionError: Bad!' }
```

A shortcut method for logging assertion results is available in the console API:

```
> console.assert(1 == 2, "Nope!")
AssertionError: Nope!
```

For a more detailed explanation of how comparison is done in JavaScript, consult: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators`.

Node's `assert` module is strongly influenced by the CommonJS test specification: `http://wiki.commonjs.org/wiki/Unit_Testing/1.0`.

Sandboxing

For some applications it is useful to be able to run a script within a protected context, isolated from the general application scope. For these purposes Node provides the `vm` module, a sandbox environment consisting of a new V8 instance and a limited execution context for running script blocks. For example, one might want to execute code submitted from a web-based editor within such a virtual machine:

```
var vm = require('vm');
var sandbox = {
  count: 2
};
var someRandomCode = 'var foo = ++count;';
vm.runInNewContext(someRandomCode, sandbox);
console.log(require('util').inspect(sandbox));
// { count: 3, foo: 3 }
```

Here we see how a provided sandbox becomes the local execution scope for the provided script. The running script can only operate within the provided sandbox object, and is denied access to even the standard Node globals, such as the running process:

```
...
var someRandomCode = 'var foo = ++count; process.exit(0);';
vm.runInNewContext(someRandomCode, sandbox);
//  vm.runInNewContext(someRandomCode, sandbox);
//     ^
//  ReferenceError: process is not defined
```

The core Node team has marked `vm` as unstable, meaning that its API and behavior may change at any given time. Two other points also need to be stated:

- This module does not guarantee a perfectly safe "jail" within which completely untrusted code can be executed. If this is your need, consider running a separate process with proper system-level permissions.
- Because `vm` spins up a new V8 instance, each invocation costs some milliseconds of startup time and about two megabytes of memory. It should be used sparingly.

For the purpose of testing code, the `vm` module can be quite effective, in particular in its ability to force code to run in a limited context. When performing a unit test, for example, one can create a special environment with mocked data simulating the environment within which the tested script will run. This can be better than creating an artificial call context with fake data. Additionally, this sandboxing will allow the execution context for new code to be better controlled, providing good protection against memory leaks and other unanticipated collisions that may bubble up while testing.

Distinguishing between local scope and execution context

Before covering further examples, we need to distinguish between the local scope of a process and its execution context. The distinction will help with understanding the difference between the two primary `vm` methods, `vm.runInThisContext` and `vm.runInNewContext`.

The execution context of a Node process represents the runtime context within V8, including native Node methods and other global objects (`process`, `console`, `setTimeout`, and so on). The members of this object are easily traced using the Node REPL:

```
node > global
```

The local scope of a Node program contains runtime definitions, such as variables defined using `var`.

A script executed through `vm.runInNewContext` has no visibility into either scope—its context is limited to the sandbox object to which it was passed, as seen earlier.

A script executed through `vm.runInThisContext` has visibility into the global execution scope, but not into the local scope. We can demonstrate this as follows:

```
var localVar = 123;
var tc = vm.runInThisContext('localVar = 321;');
console.log(localVar, tc);
var ev = eval('localVar = 321;');
console.log(localVar, ev);
//  123 321
//  321 321
```

Scripts are therefore run within contexts through `vm`. It is often useful to precompile contexts and scripts, in particular when each will be used repeatedly.

Use `vm.createContext([sandbox])` to compile an execution context, and pass in a key/value map, such as `{a: 1, b: 2}`. We'll now look at how to use these compiled objects together.

Using compiled contexts

After receiving a string of JavaScript code, the V8 compiler will do its best to optimize the code into a compiled version that runs more efficiently. This compilation step must occur each time a `vm` context method receives code as a string. If the code doesn't change and is re-used at least once, use `vm.createScript(code, [filename])` to compile it once and for all.

This Script object also inherits `runInThisContext` and `runInNewContext` methods from `vm`. Here we run a compiled script in both contexts, demonstrating how the `x` and `y` variables being incremented exist in fully isolated scopes:

```
x = 0;
y = 0;
var script = vm.createScript('++x, ++y;');
var emulation = vm.createContext({ x:0, y:0 });
for(var i=0; i < 1000; i++) {
  script.runInThisContext();
  script.runInNewContext(emulation);
}
console.log(x, y);
console.log(emulation.x, emulation.y);
//  1000  1000  1000
//  1000  1000  1000
```

Had both scripts modified the same `x` and `y` variables, then the outputs would have instead been `2000 2000`. Additionally, if the `runInNewContext` script is not passed an emulation layer (sandbox) it will throw a `ReferenceError`, having no access to the global `x` and `y` values. Try it out.

The optional `filename` argument is used for debugging, appearing in stack traces. It might help to provide one when debugging.

Errors and exceptions

The terms "error" and "exception" are often used interchangeably. Within the Node environment these two concepts are not identical. Errors and exceptions are different. Additionally, the definition of error and exception within Node does not necessarily align with similar definitions in other languages and development environments.

Conventionally, an error condition in a Node program is a non-fatal condition that should be caught and handled, seen most explicitly in the `Error as first argument` convention displayed by the typical Node callback pattern. An exception is a serious error (a system error) that a sane environment should not ignore or try to handle.

One comes across three common error contexts in Node, and should respond predictably:

- **A synchronous context**: This will normally happen in the context of a function, where a bad call signature or another non-fatal error is detected. The function should simply return an error object; `new Error(…)` or some other consistent indicator that the function call has failed.
- **An asynchronous context**: When expected to respond by firing a callback function, the execution context should pass an `Error` object, with appropriate message, as the first argument to that callback.
- **An event context**: Quoting the Node documentation: "When an EventEmitter instance experiences an error, the typical action is to emit an *error* event. Error events are treated as a special case in node. If there is no listener for it, then the default action is to print a stack trace and exit the program. " Use events where events are expected.

Clearly, these are the three situations where an error is caught in a controlled manner, prior to it destabilizing the entire application. Without falling too far into defensive coding, an effort should be made to check inputs and other sources for errors and properly dismiss them.

An additional benefit of always returning a proper `Error` object is access to the stack property of that object. The error stack shows the provenance of an error, each link in the chain of function, calls the function that led to the error. A typical `Error.stack` trace would look like this:

```
> console.log(new Error("My Error Message").stack);
Error: My Error Message
    at Object.<anonymous> (/js/errorstack.js:1:75)
    at Module._compile (module.js:449:26)
    at Object.Module._extensions..js (module.js:467:10)
    ...
```

Similarly, the stack is always available via the `console.trace` method:

```
> console.trace("The Stack Head")
Trace: The Stack Head
    at Object.<anonymous> (/js/stackhead.js:1:71)
    at Module._compile (module.js:449:26)
    at Object.Module._extensions..js (module.js:467:10)
    ...
```

It should be clear how this information aids in debugging, helping to ensure that the logical flow of our application is sound.

A normal stack trace truncates after a dozen or so levels. If longer stack traces are useful to you, try *Matt Insler's* longjohn at `https://github.com/mattinsler/longjohn`.

As well, run and examine the `js/stacktrace.js` file in your bundle for some ideas on how stack information might be used when reporting errors, or even test results.

Exception handling is different. Exceptions are unexpected or fatal errors that have destabilized the application. These should be handled with care; a system in an exception state is unstable, with indeterminate future states, and should be gracefully shut down and restarted. This is the smart thing to do.

Typically, exceptions are caught in try/catch blocks:

```
try { something.that = wontWork; }
catch(thrownError) {
  // do something with the throw
}
```

Peppering a codebase with try/catch blocks and trying to anticipate all errors can become unmanageable and unwieldy. Additionally, what if an exception you didn't anticipate—an uncaught exception—occurs? How do you pick up where you left off?

Node does not yet have a good built-in way to handle uncaught critical exceptions. This is a weakness of the platform. An exception that is uncaught will continue to bubble up through the execution stack until it hits the event loop where, like a wrench in the gears of a machine, it will take down the entire process.

The best we have is to attach an `uncaughtException` handler to the process itself:

```
process.on('uncaughtException', function(err) {
  console.log('Caught exception: ' + err);
});
setTimeout(function() {
  console.log("The exception was caught and this can run.");
}, 1000);
throwAnUncaughtException();
//  > Caught exception: ReferenceError: throwAnUncaughtException
    is not defined
//  > The exception was caught and this can run.
```

While nothing that follows our exception code will execute, the timeout will still fire, as the process managed to catch the exception, saving itself. However, this is a very clumsy way of handling exceptions.

The recently added `domain` module makes a good attempt at fixing this hole in Node's design. We will discuss the domain module next as a better tool for handling exceptions.

The domain module

Error handling in asynchronous code is also difficult to trace:

```
function f() {
  throw new error("error somwhere!")
}
setTimeout(f, 1000*Math.random());
setTimeout(f, 1000*Math.random());
```

Which function caused the error? It is difficult to say. It is also difficult to intelligently insert exception management tools. It is difficult to know what to do next. Node's `domain` module attempts to help with this and other exception localization issues. In this way code can be tested, and errors handled, with more precision.

At its simplest, `domain` sets up a context within which a function or other chunk of code can be run such that any errors occurring within that implicit domain binding will be routed to a specific domain error handler. For example:

```
var dom = domain.create();
dom.on('error', function(err) {
  console.error('error', err.stack);
});
dom.run(function() {
  throw new Error("my domain error");
});
// error Error: my domain error
//  at /js/basicdomain.js:10:8
//  ...
```

Here a function that throws an error is run within the context of a domain able to intelligently catch those exceptions, implicitly binding all event emitters, timers, and other requests created within that context.

Sometimes a method might be created elsewhere (not within the implicit context of a given `domain.run`) but is nevertheless best associated with an external domain. The `add` method exists for just such explicit binding:

```
var dom = domain.create();
dom.on("error", function(err) {
  console.log(err);
});
var somefunc = function() {
  throw new Error('Explicit bind error');
};
dom.add(somefunc);
dom.run(function() {
  somefunc();
});
// [Error: Explicit bind error]
```

Here we see how a function not implicitly bound within the `run` context can still be added to that context explicitly. To remove an execution context from a domain, use `domain.remove`. An array of all timers, functions, and other emitters added explicitly or implicitly to a domain is accessible via `domain.members`.

The way in which JavaScript's `bind` method binds a function to a context, similarly the `domain.bind` method allows an independent function to be bound to a domain:

```
var dom = domain.create();
dom.on("error", ...);
fs.readFile('somefile', dom.bind(function(err, data) {
  if(err) { throw new Error('bad file call'); }
}));
//  { [Error: bad call]
//  domain_thrown: true,
//  ...
```

Here we see how any function can be wrapped by a particular error domain "inline", a feature especially useful for managing exceptions in callbacks. Note that Error objects processed through a domain have special properties added to them:

- `error.domain`: The domain that handled the error.
- `error.domainEmitter`: If `EventEmitter` fires an error event within a domain, this function will be flagged.
- `error.domainBound`: The callback that passed an error as its first argument.

- `error.domainThrown`: A Boolean indicating whether the error was thrown or not. Errors passed to callbacks, or those announced by an EventEmitter do not trip this flag.

Another method, `domain.intercept`, functions similar to `domain.bind`, but simplifies error handling in callbacks, such that the developer will no longer need to repetitively check (or even set) the first argument of every callback `cb(err, data)` for errors. An example is found in the `js/domainintercept.js` file in your code bundle.

We may want to control when a domain is active, responsibly exiting a domain when our code is moving into a different execution stack (perhaps with its own domains), possibly re-entering the original domain at a later time. For this we use the `domain.enter` and `domain.exit` methods. Assuming we have set up two domains `dom1` and `dom2`, the first emitting `domain 1 error` and the second `domain 2 error`, we can move between domain contexts as follows:

```
dom1.add(aFuncThatThrows);
dom1.run(function() {
  dom1.exit();
  dom2.enter();
  aFuncThatThrows();
});
// domain 2 error
```

Any number of `enter` and `exit` events can be used. Note that no changes are made to the domain objects themselves, as exit does not close the domain or any such thing. If a domain needs to be destroyed you should use the `domain.dispose` method, which will also try to sanely clean up any in-flight domain I/O, aborting streams, clearing timers, ignoring callbacks, and so on.

Headless website testing with ZombieJS and Mocha

Even though we are working on the server with Node, there is no doubt that much of what we are programming will affect those connecting to our services via a browser. As it is expected that we will be using Node to test our server-side code, why not integrate browser-side testing as well?

We'll dig deeper into how to unify the entire make/build/test/deploy chain in the next section. For now, let's look into the easy to use Mocha testing framework and how some simple browser testing can be done right from the server.

The libraries we will use in this section and the next are available through the following sources:

- Mocha: `http://visionmedia.github.io/mocha/`
- Grunt: `http://www.gruntjs.com/`
- ZombieJS: `http://zombie.labnotes.org/`

PhantomJS: `http://www.phantomjs.org/`

Mocha

Mocha expects a directory containing testable units to contain a `/test` directory, within which exist various spec, or test specification and files, making up the suite of tests that Mocha will apply to your application. You will describe what you are testing, and Mocha, smartly, provides a `describe` method to do just that:

```
var assert = require("assert")
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present',
    function() {
      assert.equal(-1, [1,2,3].indexOf(5));
      assert.equal(-1, [1,2,3].indexOf(0));
    })
  })
})
```

Here we are describing a test that checks array operations. The `describe` and `it` directives are simply organizational tools, providing a title for the test suite being described (`#indexOf()`), and providing a description of each test in the suite (`should return -1 ...`). If you were to run this through Mocha, some neatly formatted test reports with these descriptions would be written to the console, and under each the pass/fail state of the test.

We have seen in the preceding example how Node's `assert` module is being used. Mocha is not an assertion library, it is a test runner that makes using an assertion library easier. You need not use Node's library; other popular assertion libraries are Chai (`http://chaijs.com/api/assert/`) and, by the maker of Mocha, Should (`https://github.com/visionmedia/should.js/`).

Fully describing Mocha is beyond the scope of this book. The preceding description should be sufficient to understand how test specifications work in the examples that will follow. You are encouraged to visit the Mocha website and run through examples yourself.

Headless web testing

One way to test if a UI is working is to pay several people to interact with a website via a browser and report any errors they find. This is expensive, inefficient, and offers incomplete coverage, as there is no guarantee that all errors will be found. Automating the process of testing a UI with browser simulation tools is the topic of this section.

A browser, stripped of its buttons and other controls, is at heart a program that validates and runs JavaScript, HTML, and CSS. That the validated HTML is rendered visually on your screen is simply a consequence of humans only being able to see with their eyes. A server can interpret the logic of compiled code and see the results of your interactions with that code without a visual component. Perhaps because eyes are usually found in one's head, a browser running on a server is typically referred to as a headless browser. We are going to learn how to do headless browser testing, first with ZombieJS (a custom-built headless browser), and later with PhantomJS (a headless version of the WebKit engine).

Most websites will use forms in some way, and a common task for developers is to validate form inputs, such that a user is warned about mismatched passwords or invalid e-mails prior to submitting the form. Consider the following form:

```
<input type="text" id="email" size="20" />
<input type="password" id="password" size="20" />
<input type="submit" value="Sign In" onclick="return
  checkForm(this.parentNode)" />
```

This is clearly a login form, requiring an e-mail address and a password. We also see that the submit button of this form will delegate validation to some JavaScript, ultimately preventing submission if the form values are invalid. How do we test that this form is correctly validating form data via Node running on a server?

Please consult the `mocha-zombie` folder in your code bundle while working through this example. To start, run `npm install` in this folder. Consult the `readme.md` file for further instructions.

Assuming that the preceding HTML file is available at `localhost:8000`, we can test it using only ZombieJS and Node's `assert` module, rolling our own testing framework :

```
var zombie = require("zombie");
var assert = require("assert");
var count = 0;
var testFramework = function(meth, v1, v2) {
  ++count;
  try {
    assert[meth](v1, v2);
    console.log("Test " + count + " OK!");
  } catch(e) {
    console.log("Test " + count + " Error!");
    console.log(e);
  }
}
browser = new zombie();
browser.visit("http://localhost:8000/login.html", function() {
  testFramework("ok", browser.success);
  testFramework("ok", browser.query('#nonexistentId'));
});
//  Test 1 OK!
//  Test 2 Error!
//  { name: 'AssertionError',
//    actual: undefined,
//    expected: true,
//    operator: '==',
//    message: '"undefined" == true' }
```

Here we see the ZombieJS object property `success` being tested (was the page found?), and a DOM query for the element with `#nonexistentId` being run on our new headless DOM, the first test succeeding and the second failing. Our mock testing framework needs to only catch assertion errors and report results.

Consult the ZombieJS website for the full set of methods available to test the DOM in much more complex ways.

Mocha is a much better testing framework than ours, so let's integrate Mocha and ZombieJS to test if our login form has both an e-mail and a password field:

```
var server = someServerRunningOn8000;
describe('Simple Login Test', function() {
  this.browser = new zombie({ site: 'http://localhost:8000' });
  this.browser.visit('/login.html');
  it('should have an input with id of #email', function() {
    assert.ok(this.browser.success);
    assert.ok(this.browser.query('input#email'));
  });
  it('should have an input with id of #password', function() {
    assert.ok(this.browser.success);
    assert.ok(this.browser.query('input#password'));
  });
});
```

After calling `mocha` on the command line, you should see something like this displayed:

```
Simple Login Test
  ✓ should have an input with id of #email
  ✓ should have an input with id of #password
```

Configure Mocha with a `/test/mocha.opts` file, an example is in your code bundle. For more information visit `http://visionmedia.github.io/mocha/#mocha.opts`.

The bundled example contains a more complex test file, but we're only scratching the surface here. There are many ways to think about and design tests, so this is only a starting point as you learn to describe your application using logical assertions, and delve deeper into Mocha and various assertion libraries.

Using Grunt, Mocha, and PhantomJS to test and deploy projects

With our new knowledge about headless browsers and test frameworks, let's develop a slightly more realistic build/test/deploy system using Node. We'll continue to use Mocha, and swap out ZombieJS for the solid PhantomJS, which unlike ZombieJS is not an emulator, it is a true WebKit browser, making it a more powerful and accurate testing environment.

We will introduce Grunt, The JavaScript Test Runner, which is designed for task automation, such as automatically running Mocha tests and deploying the current build, if all those tests pass.

The goal is to create a system that allows developers to write simple HTML pages and have Grunt to automatically test and deploy them. It is to turn this source code:

```
<head>
<!-- build:css:deploy css/style.min.css -->
    <link rel="stylesheet" href="css/normalize.css">
    <link rel="stylesheet" href="css/main.css">
<!-- /build -->
</head>
<body>
<!-- build:js:deploy js/app.min.js -->
  <script src="js/file1.js"></script>
  <script src="js/file2.js"></script>
  <script src="js/file3.js"></script>
<!-- /build -->
<!-- build:remove:deploy -->
    <script src="../../test/framework.js"></script>
    <script src="../../test/spec/file1.js"></script>
    <script src="../../test/spec/file2.js"></script>
    <script src="../../test/spec/file3.js"></script>
<!-- /build -->
</body>
```

Into this deployed code:

```
<head>
  <link rel="stylesheet" href="css/style.min.css">
</head>
<body>
  <script src="js/app.min.js"></script>
</body>
```

This is where the code testing and minification is done automatically, following the build directives surrounding script and link chunks.

At the end of this process, a new `/build` directory will be created, containing both a `stage/` and a `deploy/` folder, each containing specific builds of the files within your original `/source` directory. This is a full workflow that can help you with testing and deploying your projects.

Working with Grunt

The current workflow Grunt tries to simplify is that around Unix `make` utility. A typical `make` file to automatically run Mocha would be named `Makefile` and look something like this:

```
TESTS = test/*.js
test:
  mocha
.PHONY: test
```

It's a little cryptic, but the ideas are straightforward:

1. Set up a test runner that will run the command `mocha`.
2. Have that runner look for JavaScript spec files in the directory `test/`.

Grunt tries to accomplish the same goals within a Node environment, where calling module functions makes more sense than running system commands, and using standard JSON syntax for declaring system dependencies makes more sense.

Grunt tasks are written to a `Gruntfile.js`, and one to run Mocha tests might look like this:

```
module.exports = function(grunt) {
  grunt.initConfig({
    mocha: {
      all: ['./test/*.js']
    }
  });
  grunt.loadNpmTasks('grunt-mocha');
  grunt.registerTask('default', ['mocha']);
}
```

Adding more tasks follows the same pattern. To add the `jshint` task, which will check your JavaScript source code for possible errors, we add something like the following:

```
...
jshint: {
  options: {
    jshintrc:  ".jshintrc"
  },
  files: ["Gruntfile.js", "tasks/*.js"]
}
...
this.loadNpmTasks("grunt-contrib-jshint");
...
```

Go ahead and explore the `mocha-phantom-grunt` folder in your code bundle. You can start by installing it and running `grunt`:

```
> npm install
...
> grunt
```

Read through the output and see if you can follow what is going on. Then dive into the code and follow how Grunt sets up task runners that will ultimately validate, test, and deploy your project with ease.

Summary

The Node community has embraced testing from the beginning, and many testing frameworks and native tools are made available to developers. In this chapter we've examined why testing is so important to modern software development, as well as something about functional, unit, and integration testing, what they are and how they can be used. With the `vm` module we learned how to create special contexts for testing JavaScript programs, along the way picking up some techniques for sandboxing untrusted code.

In addition we've learned how to work with the extensive set of Node testing and error-handling tools, from more expressive console logging to Move onto one line tracing and debugging. Through the `assert` and `domain` modules we were exposed to error catching and reporting mechanisms, giving us a more robust way to specifically target errors and exceptions, rather than simply catching all in one global handler.

Finally, we learned how to set up a proper build and test system using Grunt and Mocha, in the process experimenting with two different headless browser testing libraries, learning two ways in which such testing might be done in each, and how these virtual browsers can be integrated seamlessly with other testing environments.

A
Organizing Your Work

"The way to build a complex system that works is to build it from very simple systems that work."

– Kevin Kelly

Node's straightforward module management system encourages the development of maintainable codebases. The Node developer is blessed with a rich ecosystem of clearly defined packages with consistent interfaces that are easy to combine, typically delivered via npm. When developing both simple and complex solutions, the Node developer will often find many pieces of that system readymade, and can rapidly compose those open source modules into larger, consistent, and predictable systems.

Because the value of any one module can be multiplied by adding useful submodules, the Node module ecosystem has grown rapidly.

We will cover the details of how Node understands modules and module paths, how modules are defined, how to use modules in the npm package repository, and how to create and share new npm. By following some simple rules you will find it easy to shape the structure of your application, and help others work with what you've created.

"Module" and "package" will be used interchangeably to describe the file or collection of files compiled and returned by `require()`.

Loading and using modules

The Node designers believe that most modules should be developed in userland—by developers, for developers. Such an effort is made to limit the growth of the standard library. Node's standard library contains the following short list of modules:

Network and I/O	**Strings and Buffers**	**Utilities**
TTY	Path	Utilities
UDP/Datagram	Buffer	VM
HTTP	Url	Readline
HTTPS	StringDecoder	Domain
Net	QueryString	Console
DNS		Assert
TLS/SSL		
Readline		
FileSystem		
Encryption and Compression	**Environment**	**Events and Streams**
ZLIB	Process	Child Processes
Crypto	OS	Cluster
PunyCode	Modules	Events
		Stream

Modules are loaded via the global `require` statement, which accepts the module name or path as a single argument. You are encouraged to augment the module ecosystem by creating new modules or new combinations of modules.

The module system itself is implemented in the `require` (`module`) module.

Understanding the module object

A Node module is simply a JavaScript file expected to assign a useful value to the `module.exports` property:

```
module.exports = new function() {
  this.it = function(it) {
    console.log(it);
  }
}
```

We now have a module that can be required by another file. If this module was defined in the file `./say.js`, the following is one way to use it:

```
var say = require("./say");
say.it("Hello");
// Hello
```

Note how it was not necessary to use the `.js` suffix. We'll discuss how Node resolves paths shortly.

Node's core modules are also defined using the standard `module.exports` pattern, as can be seen by browsing the source code defining `console`: `https://github.com/joyent/node/blob/master/lib/console.js`.

Once loaded, modules are cached based on their resolved filename, resolved relative to the calling module. Subsequent calls to `require(./myModule)` will return identical (cached) objects. Note however that accessing the same module via a different relative path (such as `../../myModule`) will return a different object—think of the cache being keyed by relative module paths.

A snapshot of the current cache can be fetched via `require('module')._cache)`.

The module object itself contains several useful readable properties, such as:

- **module.filename**: The name of the file defining this module.
- **module.loaded**: Whether the module is in process of loading. Boolean true if loaded.
- **module.parent**: The module that required this module, if any.
- **module.children**: The modules required by this module, if any.

You can determine if a module is being executed directly via `node module.js` or via `require('./module.js')` by checking if `require.main === module`, which will return `true` in the former case.

Resolving module paths

The `require` statement is regularly seen when browsing (or building) a Node program. You will have noticed that the argument passed to require can take many forms, such as the name of a core module or a file path.

The following pseudo code, taken from the Node documentation, is an ordered description of the steps taken when resolving module paths:

```
require(X) from module at path Y
1. If X is a core module,
   a. return the core module
   b. STOP
2. If X begins with './' or '/' or '../'
   a. LOAD_AS_FILE(Y + X)
   b. LOAD_AS_DIRECTORY(Y + X)
3. LOAD_NODE_MODULES(X, dirname(Y))
4. THROW "not found"
LOAD_AS_FILE(X)
1. If X is a file, load X as JavaScript text.  STOP
2. If X.js is a file, load X.js as JavaScript text.  STOP
3. If X.node is a file, load X.node as binary addon.  STOP
LOAD_AS_DIRECTORY(X)
1. If X/package.json is a file,
   a. Parse X/package.json, and look for "main" field.
   b. let M = X + (json main field)
   c. LOAD_AS_FILE(M)
2. If X/index.js is a file, load X/index.js as JavaScript text.  STOP
3. If X/index.node is a file, load X/index.node as binary addon.  STOP
LOAD_NODE_MODULES(X, START)
1. let DIRS=NODE_MODULES_PATHS(START)
2. for each DIR in DIRS:
   a. LOAD_AS_FILE(DIR/X)
   b. LOAD_AS_DIRECTORY(DIR/X)
NODE_MODULES_PATHS(START)
1. let PARTS = path split(START)
2. let ROOT = index of first instance of "node_modules" in PARTS, or 0
3. let I = count of PARTS - 1
4. let DIRS = []
5. while I > ROOT,
```

```
    a. if PARTS[I] = "node_modules" CONTINUE
    c. DIR = path join(PARTS[0 .. I] + "node_modules")
    b. DIRS = DIRS + DIR
    c. let I = I - 1
6. return DIRS
```

File paths may be absolute or relative. Note that local relative paths will not be implicitly resolved and must be stated. For example, if you would like to require the file `myModule.js` from the current directory it is necessary to at least prepend `./` to the file name in order to reference the current working directory—`require('myModule.js')` will not work. Node will assume you are referring to either a core module or a module found in the `./node_modules` folder. If neither exists, a `MODULE_NOT_FOUND` error will be thrown.

As seen in the pseudo code above, this `node_modules` lookup ascends a directory tree beginning from the resolved path of the calling module or file. For example, if the file at `'/user/home/sandro/project.js'` called `require('library.js')`, then Node would seek in the following order:

`/user/home/sandro/node_modules/library.js`

`/user/home/node_modules/library.js`

`/user/node_modules/library.js`

`/node_modules/library.js`

Organizing your files and/or modules into directories is always a good idea. Usefully, Node allows modules to be referenced through their containing folder, in two ways. Given a directory, Node will first try to find a `package.json` file in that directory, alternatively seeking for an `index.js` file. We will discuss the use of the `package.json` files in the next section. Here we need to simply point out that if `require` is passed the directory `./myModule` it will look for is `./myModule/index.js`.

If you've set the `NODE_PATH` environment variable then Node will use that path information to do further searches if a requested module is not found via normal channels. For historical reasons `$HOME/.node_modules`, `$HOME/.node_libraries`, and `$PREFIX/lib/node` will also be searched. `$HOME` represents a user's home directory, and `$PREFIX` will normally be the location Node was installed to.

Using npm

As mentioned when discussing how Node does path lookups, modules may be contained within a folder. If you are developing a program as a module for others to use you should bundle that module within its own folder and publish it. The npm package management system is designed to help you do just that.

As we've seen throughout the examples in this book, a `package.json` file describes a module, usefully documenting the module's name, version number, dependencies, and so forth. It must exist if you would like to publish your package via npm. In this section we will investigate the key properties of this file, and offer some hints and tips on how to configure and distribute your modules.

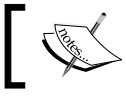
Try `npm help json` to fetch detailed documentation for all the available `package.json` fields, or visit `https://npmjs.org/doc/json.html`.

A `package.json` file must conform to the JSON specification. For example, properties and values must be double-quoted.

Initializing a package file

You can create a package file by hand, or use the handy `npm init` command-line tool, which will prompt you for a series of values and automatically generate a `package.json` file. Let's run through some of these values:

- **name**: (Required) This string is what will be passed to `require()` in order to load your module. Make it short and descriptive, using only alphanumeric characters—this name will be used in URLs, command line arguments, and folder names. Try to avoid using "js" or "node" in the name.
- **version**: (Required) npm uses semantic versioning, where these are all valid:
  - >=1.0.2 <2.1.2
  - 2.1.x
  - ~1.2



For more information on version numbers, visit `https://npmjs.org/doc/misc/semver.html`.

- **description**: When people search `npmjs.org` for packages, this is what they will read. Make it short and descriptive.

- **entry point**: This is the file that should set `module.exports`—it defines where the module object definition resides.
- **keywords**: A comma-separated list of keywords that will help others find your module in the registry.
- **license**: Node is an open community that likes permissive licenses. "MIT" and "BSD" are good ones here.

You might also want to set the `private` field to `true` while you are developing your module. This ensures that npm will refuse to publish it, avoiding accidental releases of incomplete or time-sensitive code.

Using scripts

npm is ultimately a build tool, and the `scripts` field in your package file allows you to set various build directives executed at some point following certain `npm` commands. For example, you might want to minify JavaScript, or execute some other processes that build dependencies that your module will need whenever `npm install` is executed. The available directives are:

- prepublish, publish, postpublish: Run by the `npm publish` command.
- preinstall, install, postinstall: Run by the `npm install` command.
- preuninstall, uninstall, postuninstall: Run by the `npm uninstall` command.
- preupdate, update, postupdate: Run by the `npm update` command.
- pretest, test, posttest: Run by the `npm test` command.
- prestop, stop, poststop: Run by the `npm stop` command.
- prestart, start, poststart: Run by the `npm start` command.
- prerestart, restart, postrestart: Run by the `npm restart` command. Note that `npm restart` will run the `stop` and `start` scripts if no `restart` script is provided.

It should be clear that the commands with prefix pre will run before, and the commands with prefix post will run after their primary command (such as `publish`) is executed.

Package files demonstrating the use of script directives can be found in the `mocha-zombie` folder in the code bundle for *Chapter 9, Testing Your Application*.

Declaring dependencies

It is likely that a given module will itself depend on other modules. These dependencies are declared within a `package.json` file using four related properties, which are:

- **dependencies**: The core dependencies of your module should reside here.
- **devDependencies**: Some modules are only needed during development (and not in production), such as those used for testing. Declare those modules here.

npm install will always install both `dependencies` and `devDependencies`. To install an npm package without loading `devDependencies`, use the command `npm install --production`.

- **bundledDependencies**: Node is changing rapidly, as are npm packages. You may want to lock a certain bundle of dependencies into a single bundled file and have those published with your package, such that they will not change via the normal `npm update` process.
- **optionalDependencies**: Contains modules that are optional. If these modules cannot be found or installed the build process will not stop (as it will with other dependency load failures). You can then check for this module's existence in your application code.

Dependencies are normally defined with a npm package name followed by versioning information:

```
"dependencies" : {
  "express" : "3.3.5"
}
```

However they can also point to a tarball:

```
"foo" : "http://foo.com/foo.tar.gz"
```

You can point to a GitHub repository:

```
"herder": "git://github.com/sandro-pasquali/herder.git#master"
```

Or even the shortcut:

```
"herder": "sandro-pasquali/herder"
```

These GitHub paths are also available to `npm install`. For example, `npm install sandro-pasquali/herder`.

Additionally, in cases where only those with proper authentication are able to install a module, the following format can be used to source secure repositories:

```
"dependencies": {
  "a-private-repo":
    "git+ssh://git@github.com:user/repo.git#master"
}
```

By properly organizing your dependencies by type, and intelligently sourcing those dependencies, build requirements should be easy to accommodate using Node's package system.

Publishing packages

In order to publish to npm, you will need to create a user. `npm adduser` will trigger a series of prompts requesting your name, e-mail, and password. You may then use this command on multiple machines to authorize the same user account.

To reset your npm password visit `https://npmjs.org/forgot`.

Once you have authenticated with npm you will be able to publish your packages using the command `npm publish`. The easiest path is to run this command from your package folder. You may also target another folder for publishing (remember that a `package.json` file must exist in that folder).

You may also publish a gzipped TAR archive containing a properly configured package folder.

Note that if the `version` field of the current `package.json` file is lower or equal to that of the existing, published package npm will complain and refuse to publish. You can override this by using the `--force` argument with `publish`, but you probably want to update the version and republish.

To remove a package use `npm unpublish <name>[@<version>]`. Note that once a package is published other developers may depend on it. For this reason you are strongly discouraged from removing packages that others are using. If what you want to do is discourage the use of a version, use `npm deprecate <name>[@<version>] <message>`.

To further assist collaboration, npm allows multiple owners to be set for a package:

- `npm owner ls <package name>`: Lists the users with access to a module.
- `npm owner add <user> <package name>`: The added owner will have full access, including the ability to modify the package and add other owners.
- `npm owner rm <user> <package name>`: Removes an owner and immediately revokes all privileges.

All owners have equal privileges—special access controls are unavailable, such as being able to give write but not delete access.

Globally installing packages and binaries

Some Node modules are useful as command-line programs. Rather than requiring something such as `> node module.js` to run a program, we might want to simply type `> module` on the console and have the program execute. In other words, we might want to treat a module as an executable file installed on the system `$PATH` and therefore accessible from anywhere. There are two ways to achieve this using npm.

The first and simplest way is to install a package using the `-g (global)` argument:

```
npm install -g module
```

If a package is intended as a command-line application that should be installed globally, it is a good idea to set the `preferGlobal` property of your `package.json` file to `true`. The module will still install locally, but users will be warned about its global intentions.

Another way to ensure global access is by setting a package's `bin` property:

```
"name": "aModule",
"bin" : {
  "aModule" : "./path/to/program"
}
```

When this module is installed, `aModule` will be understood as a global CLI command. Any number of such programs may be mapped to `bin`. As a shortcut, a single program can be mapped like so:

```
"name": "aModule",
"bin" : "./path/to/program"
```

In this case the name of the package itself (`aModule`) will be understood as the active command.

Sharing repositories

Node modules are often stored in version control systems, allowing several developers to manage package code. For this reason the `repository` field of `package.json` can be used to point developers to such a repository, should collaboration be desired. For example:

```
"repository" : {
  "type" : "git",
  "url" : "http://github.com/sandro-pasquali/herder.git"
}
"repository" : {
  "type" : "svn",
  "url" : "http://v8.googlecode.com/svn/trunk/"
}
```

Similarly, you might want to point users to where bug reports should be filed by using the bugs field:

```
"bugs": {
  "url": "https://github.com/sandro-pasquali/herder/issues"
}
```

B
Introducing the Path Framework

"*Great acts are made up of small deeds.*"

– Lao Tzu

JavaScript is the only language natively installed in browsers. All web-based applications depend on JavaScript to deliver their UI. If you are using a framework built on another language (such as Ruby on Rails), you must merge two completely different development models, one for the client and one for the server. With the arrival of Node and JavaScript on the server, this unfortunate and awkward workflow has been made smooth and elegant.

The Path Framework is a powerful full-stack application development platform that will help you create enterprise-class applications in less time. Unlike Express, which is limited to the server side of an application, and client-side frameworks such as EmberJS that offer no help on building servers, Path unifies your application development process. With its focus on the full stack, Path enables the JavaScript developer to build complete applications using a single language that are scalable, easy to maintain, test, and modify. Of course, Node occupies a central place in the Path design.

Path's goal is to help developers easily build web applications that work like native applications. This means responsive single-page applications with powerful features.

Data is synchronized automatically between client and server when you build an application with Path. No more building complex and error-prone client/server synchronization hooks. Client interfaces are composed out of modules of any number and complexity, easily using an elegant, declarative, and data-centric syntax. Real-time data broadcasts across groups of clients, in both directions, in a snap. Programming route handlers on the server is just as easy as binding a click to a button.

Some of the key features of Path are as follows:

- **Just JavaScript**: Context switching is hard. Finally, a unified team speaking a common language, across the stack.
- **Automatic UI refresh**: Your UI modules automatically update when the data they are bound to changes. Two-way data binding without the knots.
- **No more CRUD**: Move beyond the dirty old CRUD model into an efficient socket-based communication paradigm designed for modern real-time application development.
- **Hot code updates**: Change your code on the server and the clients will be updated automatically.
- **Interoperability**: Anything that can communicate over WebSockets can communicate with a Path server.
- **Dirt simple module loading**: Simple declarative HTML syntax helps you compose UIs in a way that is easy for you to create and easy for others to read.
- **Automatic module caching**: It means smart dependency loading.
- **As easy as HTML**: Modules are naturally expressed by nesting HTML elements. Loaded modules can themselves create further module dependency trees. Element attributes store options and other component settings. User events are bound to server actions without using any JavaScript.
- **Embeddable**: Use as much or as little of Path as you'd like. Path is here to help, not hinder. Other frameworks can even share your Path server.
- **Out-of-the-box productivity**: Path comes packed with prebuilt applications you can get started with right away.

To get started, clone the Path repository from `https://github.com/sandro-pasquali/pathjs.git`, and follow the instructions on that page.

Managing state

As we discussed earlier:

If changes can be introduced into a web application without requiring a complete reconstruction of state and state display then updating client information becomes cheaper. The client and server can talk more often, regularly exchanging information. Servers can recognize, remember, and respond immediately to client desires, aided by reactive interfaces gathering user actions and reflecting the impact of those actions within a UI in near real time.

A very important question every application developer must ask is this one: where should state reside?

Of course, permanent data must ultimately reside in a persistent layer such as a server-side database. As well, transient-state data (such as which navigation item is highlighted in a UI) can happily exist in the client. However, if an application displays real bank records in a data grid within a browser, how do changes made to the canonical record (the database) synchronize with changes in client model?

For example, imagine a banking application where a browser-based UI indicates to the user that she has a balance of USD 100. This UI was "drawn" based on an earlier request to a server, asking something like: What is Mary's bank balance? Clearly that number may change in the next second, or even millisecond, given the automated nature of banking. How can Mary be sure that when she pays a USD 100 bill she still has USD 100 in the bank?

One solution is to periodically expire the validity of the client state and poll a server for state changes, resynching client and server state. Additionally, client requests are revalidated on the server—Mary would be warned if she tries to spend USD 100 that she doesn't have. Client state is *optimistic*, and transactions are *revalidated*.

However, there are several problems with this model:

- **Not all state can exist on a client**: Secured state data must remain on the server, which means clients can handle only pieces of state, making even the best attempt to maintain a synchronized client state imperfect by definition.
- **Revalidation is not an acceptable solution in many cases**: Consider a real-time shooter. If one player alters the state of another player (let's say by killing that player), it is not acceptable to at some point in the future reanimate the killed player simply because the shooter's client had incorrect position information that renders the original kill invalid.

- **Optimism leads to false security**: *Meaningless* is one word that can be used to described state data that may or may not be valid. For example, a stock-trading program showing the wrong price for FOO Corporation may lead a trader to short sell BAR Corporation. If the state of BAR (according to the client) matches that on a server the trade will go through, and the trader may lose money. Any single bit of information cannot itself validate all other bits of information. Indeterminacy is a feature of concurrent systems, and these systems challenge *deductive* state validation techniques—whether a comparison is valid cannot always be determined.

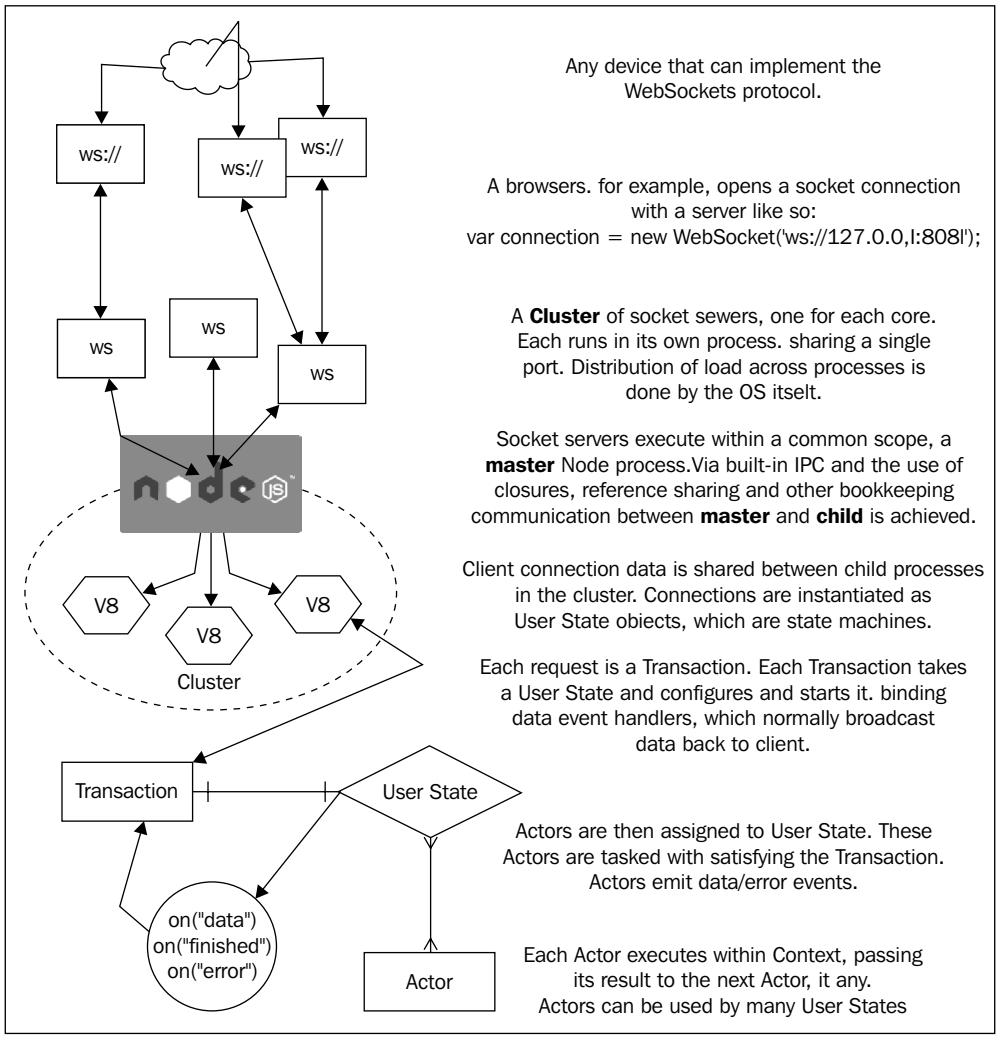
Many techniques have been used to synchronize client and server state—cookies, hidden form fields, fat URLs, and others. Path takes a different view, based on the advantages that Node provides, in particular when paired with the WebSocket protocol. If we can create highly responsive servers linked with clients through efficient pipelines, where sub-second response times are expected, it is better to simply keep important state information *on the server* and cease any attempts to create client-side reflections on a mirror that is easily smudged.

Bridging the client/server divide

Node is designed for high-concurrency environments, where many clients connecting simultaneously can have their needs met quickly and predictably. This ability also implies that each client can expect a Node server to respond rapidly when it itself sends many simultaneous requests.

One of Path's key design beliefs is that keeping state synchronized across clients and servers is complex and difficult to do well. For this reason, applications built with Path exchange data between clients and servers exclusively through the WebSockets protocol.

The following is a high-level view of how Path handles client requests. We see how the `cluster` module is used to share responsibility for handling socket connections among several socket servers:



Once a request has been assigned to a server that request must be fulfilled. We will call that a transaction. Each transaction is being fulfilled for a single user and, as such, fulfillment is delegated to a user object. This user object is a state machine, containing any specific state information for this user. Depending on transaction fulfillment requirements, one or more actors will be assigned, emitting their results (or error warnings). Authenticated users will have a unique machine to process their requests, while other users will share a common "guest" machine.

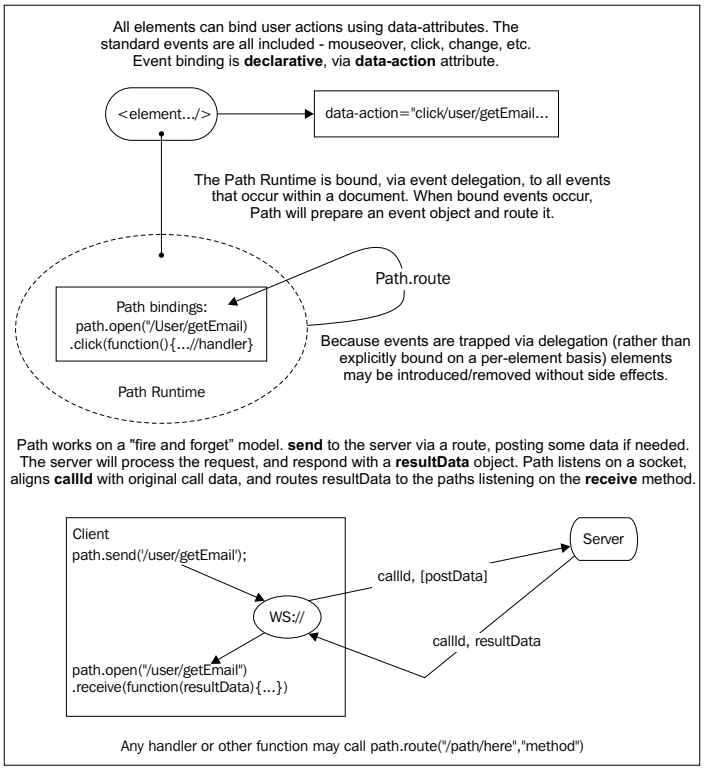
In this way each request can be framed within a user scope, accurately resolved within the *canonical* state and responded to with confidence.

Sending and receiving

In addition to keeping state on the server, Path also aims at reducing the responsibility of a caller for holding on to a call context until a response is received. This sort of call frame maintenance anticipating callback execution is an essential condition of nearly all AJAX development patterns, and is typically represented as follows:

```
//Within some execution context, such as an autocomplete input
someXhrProxy.get("/a/path/", function(data) {
  //A callback bound to the execution context via closures
});
```

Some client-side libraries attempt to simplify this pattern with abstractions like Promises, but they miss the point: a call should "fire and forget", its job being solely the transmission of a request. The impact of that action, for both the server and the client, is not the caller's concern. A developer only needs to assert a desired change of state, or request some information. Path facilitates this separation of concerns, absolves the call function of any responsibility for maintaining call contexts at the *functional* level and manages the flow of execution itself:



Path uses a declarative model for binding user actions. For example, the following is all that is necessary to bind a `click` event to an element:

```
a href="#" data-action="click/create/user/jack">Create user</a>
```

We create the action to execute when that element is clicked by opening the set path, and listening for the `click` event:

```
path
.open("/create/user/:username")
.click(function(username) {
  console.log("New user: " + username);
})
```

Clicking on the element will log `New user: jack` to the console.

To send a request to the server, we follow a similar pattern. If, for example, we wanted to route the `/create/user/jack` path to a server, creating a new user, we would use the `send` command:

```
path
.open("/create/user/:username")
.click(function(username) {
  path.send("/create/user/" + username);
})
```

See the examples in your code bundle for more involved transactions.

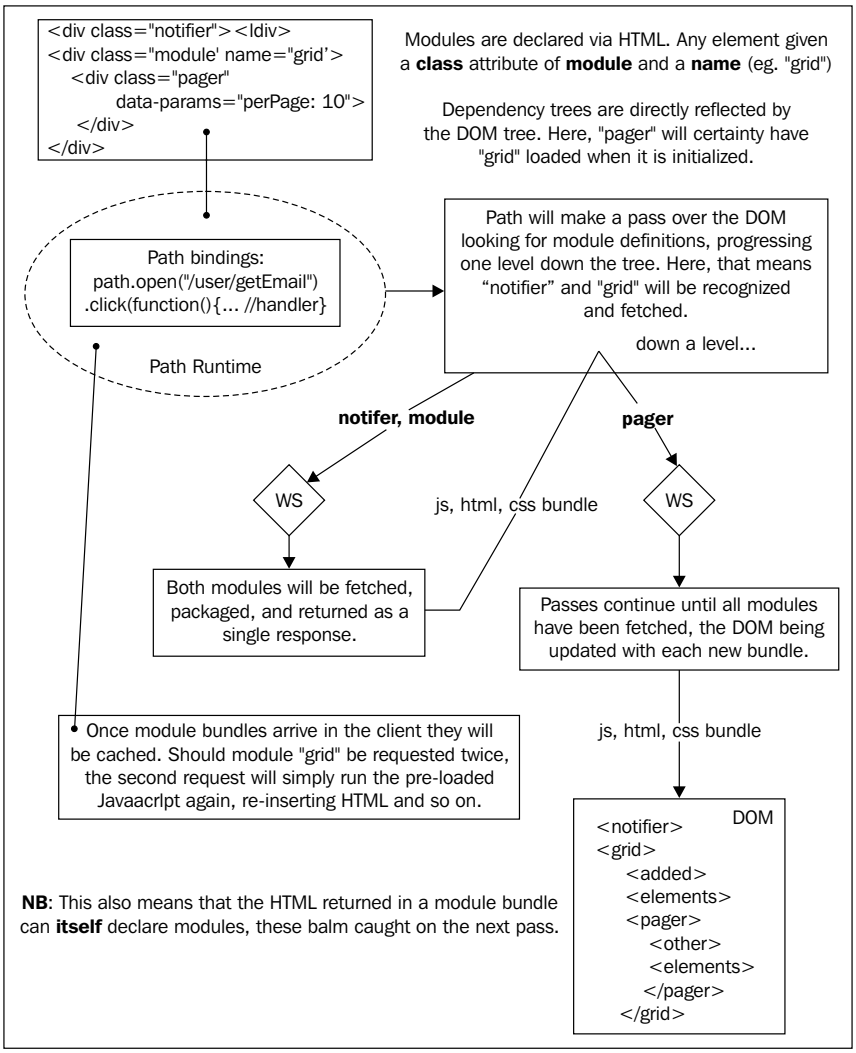
Achieving a modular architecture

Path's declarative model also makes it very easy to load modules and dependencies via HTML. Some of the key goals of its module loading system are as follows:

- **Bundling**: When it is known that a group of modules is needed, bundle them into one package and send them in one request.
- **Caching**: Once a module has been retrieved from the server, it should not need to be requested again. Path maintains a cache of module data on the client.
- **Communication**: Path routes events through paths such as `/click/get/email`. Modules do the same, broadcasting their data along `module/moduleName/eventName` paths, allowing listeners to act on those transmissions.
- **Encapsulation**: No direct access to a module's internals is allowed. All communication is via paths, as described previously.

- **Templates and data binding**: Each module is easily bound to a data source, served through a real time socket server. When new data arrives, the module will automatically update its view.

Path was designed to help simplify the designing of modular architectures. Through a declarative syntax, Path makes it easy to see how an application is laid out, even for a nontechnical person. Additionally, the natural binding mechanism for modules further simplifies comprehension of "what is going on", as well as aiding the creation of real-time UI updates, as data bindings automatically receive socket updates, piping new data into new view templates as it arrives:



For example, if I have a list of items to display I might use a special module named `list`. A list module may want tool tips to appear when items in its list are hovered over, so it may depend on a `tooltip` dependency. This relationship would be expressed as follows:

```
<div class="module" name="list">
  <div class="module" name="tooltip"></div>
  <ul>
    <li>...</li>
    <li>...</li>
  </ul>
</div>
```

The `list` module may now use `tooltip` features. Another way of expressing this relationship is as follows:

```
<div class="module" name="tooltip"></div>
<div class="module" name="list">
  <ul>
    <li>...</li>
    <li>...</li>
  </ul>
</div>
```

The advantage of the second method is that, while continuing to ensure that `tooltip` exists when list is loaded, both modules can be loaded in parallel, as they exist on the same level within a DOM tree.

This built-in tendency towards facilitating bundling and parallel loading helps applications developed with Path remain performant, regardless of number of modules or the complexity of dependencies. The flexibility provided here by Path makes dependency loading and assignment transparent and efficient.

Automatically binding is also easy to achieve. Consider the following HTML:

```
<textarea data-binding="form.something.selected"></textarea>
<select data-binding="form.something.selected">
  <option value="one">1</option>
  <option value="two">2</option>
</select>
```

Note how both the `select` element and the `textarea` element have identical data-binding properties. Path will automatically "wire up" this relationship such that whenever a new selection is made the content of the given `textarea` will reflect the new selected data. With very little work, very complex data relationships and interactive feedback loops can be created using Path.

Investigate the examples given in your code bundle, and play around with different orderings and nestings of modules, different data bindings. **With Path we can realistically begin to design applications in which a client is understood as simply another event emitter and event listener, not unlike a `Stream` or `process.stdin`, the same holds true for the client's perception of the server.** With its full-stack comprehensive design, Path is the real-time framework designed for the Node developer.

C

Creating your own C++ Add-ons

> "*If two (people) on the same job agree all the time then one is useless. If they disagree all the time, then both are useless.*"
>
> *– Darryl F. Zanuck*

A very common description of Node is this: Node.js allows JavaScript to run on the server. While true, this is also misleading. Node's creators organized and linked powerful C++ libraries in such a way that their efficiency could be harnessed without developers needing to comprehend their complexities. Node abstracts away the complexity of multiuser, simultaneous multithreaded I/O by wrapping that concurrency model into a single-threaded environment that is easy to understand and already well understood by millions of web developers.

When you are working with Node you are ultimately working with C++, a language whose suitability for enterprise-level software development no one would question.

Node's C++ foundation puts lie to claims that Node is not enterprise-ready. These claims confuse what JavaScript's role in the Node stack actually is. The bindings to Redis and other database drivers used regularly in Node programs are C bindings–fast, near the "metal". Node's process bindings (`spawn`, `exec`, and so on) facilitate a smooth integration of powerful system libraries (such as ImageMagick) with headless browsers and HTTP data streams. With Node, we are able to access the enormously powerful suite of native Unix programs through the ease and comfort of JavaScript. And of course, we can write our own C++ add-ons.

Paraphrasing Professor Keith Devlin's description in *Calculus: One of the Most Successful Technologies* (`https://www.youtube.com/watch?v=8ZLC0egL6pc`), these are some features of successful consumer technologies:

- It should remove difficulty or drudgery from the process of completing a task.
- It should be easy to learn and use.
- It should be easier to learn and use than the popular method, if one exists.
- Once learned, it can be used without constant expert guidance. A user remains able to remember and/or derive most or all of the rules governing interactions with the technology through time.
- It should be possible to use it without knowing how it works.

Hopefully, as you think about the class of problems Node aims to solve, and the form of the solution it provides, the above five features are easily seen in the technology Node represents. Node is fun to learn and use, with a consistent and predictable interface. Importantly, "under the hood" Node runs enormously powerful tools that the developer only needs to understand in terms of their API.

Wonderfully, Node, V8, libuv, and the other libraries composing the Node stack are open sourced, a significant fact that further distinguishes Node from many competitors. Not only can one contribute directly to the core libraries but one can cut and paste code blocks and other routines to use in one's own work. You should see your growth into a better Node developer as a chance to simultaneously become a better C++ programmer.

This is not a primer on C++, leaving you to pursue this study on your own. Don't be intimidated! The C-family of languages is designed using forms and idioms not unlike what you are already used to with JavaScript. For example, `if..then` constructs, `for` loops, functions, and returning values from functions, even semicolon-terminated lines, are central to JavaScript and regularly used in C++ programming. As a JavaScript developer you should be able to understand the design and goals of the following examples with little effort, and can dip into C++ programming to resolve the meaning of the parts that aren't clear. Extending these examples iteratively is an excellent way to gently enter the world of C++ programming.

Hello World

Let's build our first add-on. In keeping with tradition, this add-on will result in a Node module that will print out `Hello World!`. Even though this is a very simple example, it typifies the structure of all further C++ add-ons you will build. This allows you to incrementally experiment with new commands and structures, growing your knowledge in easy to understand steps.

C++ files are typically given a `.cc` extension. To start, we create a `hello.cc` file:

```
#include <node.h>
#include <v8.h>

using namespace v8;

Handle<Value> Method(const Arguments& args) {
  HandleScope scope;
  return scope.Close(String::New("Hello World!"));
}

void init(Handle<Object> target) {
  target->Set(String::NewSymbol("hello"),
    FunctionTemplate::New(Method)->GetFunction());
}

NODE_MODULE(hello, init)
```

This example includes the Node and V8 libraries, building our module using the same libraries Node modules build on `NODE_MODULE` is simply a macro that greatly simplifies the export of an initialization function, which every add-on should provide. The name of the module itself should be passed as a first argument. The second argument should point to the initializing function.

You are embedding C++ code into the V8 runtime such that JavaScript can be bound into the relevant scope. V8 must scope all the new allocations made in your code, and so you'll need to wrap the code you write, extending V8. To this end you'll see several instances of the `Handle<Value>` syntax, wrapping C++ code in the examples that follow. Comparing these wrappers to what will be defined in the initialization function pushed out to `NODE_MODULE` should make it clear how Node is being bound to C++ methods via the V8 bridge.

To learn more about how V8 embeds C++ code, visit `https://developers.google.com/v8/embed`.

Unlike JavaScript, C++ must be compiled prior to executing. This compile step converts human-readable code into optimized bytecode. To compile C++ code such that it can be easily used as a Node add-on, we will need to use `node-gyp`, the tool necessary for building out our add-ons. To install it, simply run `npm install -g node-gyp`.

The benefit of using `node-gyp` (vs. native compilers like `gcc`) is that it removes the pain of dealing with the differences between platforms. You can just write code and know that the build process will work. To that end, let's compile our `hello.cc` file.

The first step is to create a `binding.gyp` file, which will provide the compiler with necessary build information. A basic file will suffice in our case:

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

Note the `target_name` of `"hello"`, corresponding to the `target->Set(String::NewSymbol("hello")` line in our `.cc` file, and the linking of our source `hello.cc` file.

In cases where more than one source file needs to be compiled, simply add it to the `sources` array.

Within the same directory, enter `node-gyp configure` into your terminal. You will see information about the build process printed to your terminal. Once completed a new `build/` directory will exist, containing various files. Importantly, there will exist a `Makefile`. Go ahead and change current directory into `build/` and run `make`. You will see something like this in your terminal:

```
CXX(target) Release/obj.target/hello/hello.o
SOLINK_MODULE(target) Release/hello.node
SOLINK_MODULE(target) Release/hello.node: Finished
```

The build process is now complete. As described in the preceding output, a `Release/` directory has been created containing a compiled `hello.node` file. This is the file we will be linking to within our Node code.

As a shortcut you can use `node-gyp configure build` to configure and build in one step.

Once we've built and compiled our add-on, we can begin using it immediately within our own Node programs. Jump back into the directory containing `hello.cc`. Within this directory create the follow Node program:

```
var addon = require('./build/Release/hello');
console.log(addon.hello());
```

Run this program, and you will see `Hello World!` displayed in your terminal. You are now a C++ programmer!

Creating a calculator

Of course one would never bother to write an add-on to simply echo back strings. It is more likely that you will want to expose an API or interface to your Node programs. Let's create a simple calculator, with two methods: `add` and `subtract`. In this example, we will demonstrate how to pass arguments from JavaScript to methods within an add-on, and to send any results back.

The complete code for this example will be found in your code bundle. The meat of the program can be seen in this snippet, where we define an interface for our two methods, each one expect to receive two numbers as arguments:

```
#include <node.h>
#include <v8.h>

using namespace v8;

Handle<Value> Add(const Arguments& args) {
  HandleScope scope;

  if(args.Length() < 2) {
    ThrowException(Exception::TypeError(String::New("Wrong number
      of arguments")));
    return scope.Close(Undefined());
  }

  if(!args[0]->IsNumber() || !args[1]->IsNumber()) {
    ThrowException(Exception::TypeError(String::New("Wrong
      arguments")));
    return scope.Close(Undefined());
  }
```

```
  Local<Number> num = Number::New(args[0]->NumberValue() +
    args[1]->NumberValue());
  return scope.Close(num);
}

Handle<Value> Subtract(const Arguments& args) {
  // Similar method to do subtraction...
}

void Init(Handle<Object> exports) {
  exports->Set(String::NewSymbol("add"),
    FunctionTemplate::New(Add)->GetFunction());
  exports->Set(String::NewSymbol("subtract"),
    FunctionTemplate::New(Subtract)->GetFunction());
}

NODE_MODULE(calculator, Init)
```

We can quickly see that two methods have been scoped: `Add` and `Subtract` (`Subtract` is defined nearly identically with `Add`, with only a change of operator). Within the `Add` method we see an `Arguments` object (reminiscent of JavaScript's `arguments` object) that is checked for length (we expect two arguments) and type (we want numbers). Take a good look at how this method closes out:

```
Local<Number> num = Number::New(args[0]->NumberValue() + args[1]-
  >NumberValue());
return scope.Close(num);
```

While there seems to be a lot going on, it is really rather simple: V8 is instructed to allocate space for a `Number` variable with name `num`, to be assigned the value of adding our two numbers together. When this operation has been completed, we close out the execution scope and return `num`. We don't have to worry about memory management for this reference, as that is automatically handled by V8.

Finally, we see in the following chunk not only how this particular program defines its interface, but how, at a deep level, Node modules and the `exports` object are in fact associated:

```
void Init(Handle<Object> exports) {
  exports->Set(String::NewSymbol("add"),
    FunctionTemplate::New(Add)->GetFunction());
  exports->Set(String::NewSymbol("subtract"),
    FunctionTemplate::New(Subtract)->GetFunction());
}
```

As in our `"hello"` example, here we see the new symbols (these are just types of strings) `add` and `subtract`, which represent the method names for our new Node module. Their function is implemented with the easy-to-follow `FunctionTemplate::New(Add)->GetFunction())`.

Using our calculator from a Node program is now rather easy:

```
var calculator = require('./build/Release/calculator');

console.log(calculator.add(2,3));
console.log(calculator.subtract(3,2));
```

When this is executed you will see the following displayed in your terminal:

```
5
1
```

Implementing callbacks

In keeping with the typical pattern of a Node program, add-ons also implement the notion of callbacks. As one might expect in a Node program, a C++ add-on performing an expensive and time-consuming operation should comprehend the notion of asynchronously executing functions.

The following code will expose a method that will pass back the current system time to any callback it is sent:

```
#include <node.h>
#include <ctime>

using namespace v8;

Handle<Value> GetTime(const Arguments& args) {
  HandleScope scope;

  Local<Function> cb = Local<Function>::Cast(args[0]);
  const unsigned argc = 1;
  time_t stamp = time(0);
  Local<Value> argv[argc] = {
    Local<Value>::New(Number::New(stamp))
  };
  cb->Call(Context::GetCurrent()->Global(), argc, argv);

  return scope.Close(Undefined());
}
```

```
void Init(Handle<Object> exports, Handle<Object> module) {
  module->Set(String::NewSymbol("exports"),
    FunctionTemplate::New(GetTime)->GetFunction());
}

NODE_MODULE(callback, Init)
```

Here we include the `ctime` standard library, using its time method when setting up our `GetTime` handle. The magic happens within this handle, where your function argument is properly bound and called.

Once you have created your binding file and compiled everything, go ahead and run your callback module:

```
var timeNow = require('./build/Release/callback');

timeNow(function(stamp){
  console.log(stamp);
});
//  1481315296
```

Beginning with these tools you can start developing some equally simple add-ons, eventually moving into deeper territory.

Closing thoughts

Being able to easily link C++ modules with your Node program is a powerful new paradigm. It may be tempting, then, to exuberantly begin writing C++ add-ons for every identifiable segment of your programs. While this might be a productive way to learn, it is not necessarily the best idea in the long run. While it is certainly true that in general, compiled C++ will run more quickly than JavaScript code, remember that V8 is ultimately using another type of compilation on the JavaScript code it is running. JavaScript running within V8 runs very efficiently. As well, we don't want to lose the simplicity of organization and predictable single-threaded runtime of JavaScript when designing complex interactions within a high-concurrency environment. Remember: Node came into being partly as an attempt to save the developer from working with threads and related complexities when performing I/O. As such, try and keep these two rules in mind:

- **Is a C++ module actually going to run more quickly?**: The answer isn't always yes. The extra step of jumping into a different execution context and then back into V8 is wasteful if your add-on is only returning a static string. Felix Geisendorfer's talk describing his work with building fast MySQL bindings provides some insight into these decisions: `http://www.youtube.com/watch?v=Kdwwvps4J9A`.

- **How does splitting up your codebase affect maintainability?**: While it would be hard for any developer to suggest using less-efficient code, sometimes a negligible performance gain does not overcome an increase in complexity that can lead to harder-to-find bugs or difficulties with sharing or otherwise managing your codebase.

Node merged a beautiful JavaScript API with an enormously powerful and easily extensible application stack. Given the ability to integrate C++ into your applications, there is no reason to exclude Node from the list of technologies to consider for your next project.

Links and resources

The Node documentation for add-ons is excellent: `http://nodejs.org/api/addons.html`.

A repository containing many examples of add-ons can be found here: `https://github.com/rvagg/node-addon-examples`.

An excellent resource for those learning C++: `http://www.learncpp.com/`.

When you're feeling more confident, the source code for Node's core modules is an excellent place to both explore and to learn from: `https://github.com/joyent/node/tree/master/src`.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Thank you for buying

Mastering Node.js

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

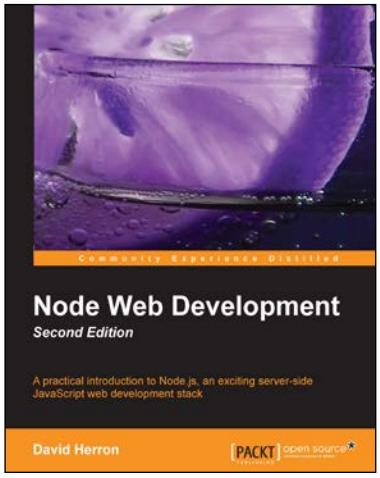
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

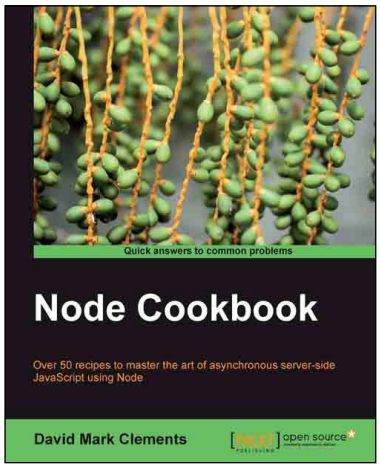


Node Web Development - Second Edition

ISBN: 978-1-78216-330-5 Paperback: 248 pages

A practical introduction to Node.js, an exciting server-side JavaScript web development stack

1. Learn about server-side JavaScript with Node.js and Node modules.
2. Website development both with and without the Connect/Express web application framework.
3. Developing both HTTP server and client applications.



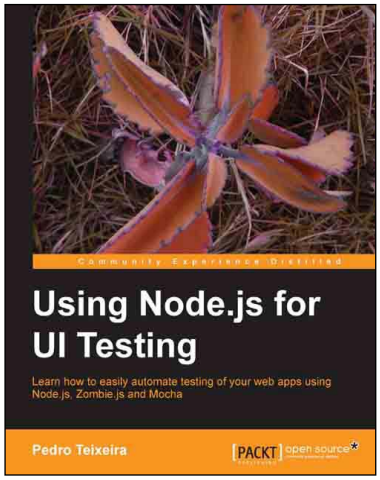
Node Cookbook

ISBN: 978-1-84951-718-8 Paperback: 342 pages

Over 50 recipes to master the art of asynchronous server-side JavaScript using Node

1. Packed with practical recipes taking you from the basics to extending Node with your own modules
2. Create your own web server to see Node's features in action
3. Work with JSON, XML, web sockets, and make the most of asynchronous programming

[PACKT] open source
community experience distilled
PUBLISHING



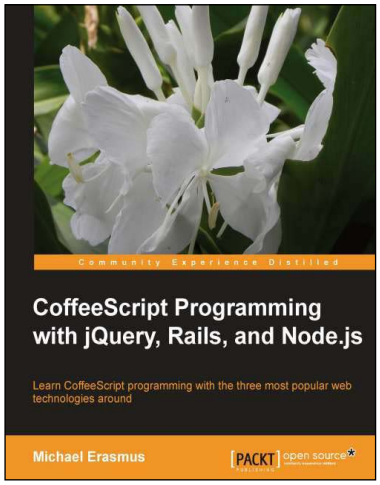
Using Node.js for UI Testing

ISBN: 978-1-78216-052-6 Paperback: 146 pages

Learn how to easily automate testing of your web apps using Node.js, Zombie.js and Mocha

1. Use automated tests to keep your web app rock solid and bug-free while you code
2. Use a headless browser to quickly test your web application every time you make a small change to it
3. Use Mocha to describe and test the capabilities of your web app

---



CoffeeScript Programming with jQuery, Rails, and Node.js

ISBN: 978-1-84951-958-8 Paperback: 140 pages

Learn CoffeeScript programming with the three most popular web technologies around

1. Learn CoffeeScript, a small and elegant language that compiles to JavaScript and will make your life as a web developer better
2. Explore the syntax of the language and see how it improves and enhances JavaScript
3. Build three example applications in CoffeeScript step by step