



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Node Web Development

Second Edition

A practical introduction to Node.js, an exciting server-side JavaScript web development stack

David Herron

[PACKT] open source*
PUBLISHING community experience distilled

Node Web Development

Second Edition

A practical introduction to Node.js, an exciting
server-side JavaScript web development stack

David Herron

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

Node Web Development

Second Edition

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2011

Second Edition: July 2013

Production Reference: 1120713

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

ISBN 978-1-78216-330-5

www.packtpub.com

Cover Image by Suresh Mogre (suresh.mogre.99@gmail.com)

Credits

Author

David Herron

Reviewers

Ollie Bennett

Nathan Zondlo

Acquisition Editors

Joanne Fitzpatrick

Sarah Cullington

Lead Technical Editor

Mayur Hule

Sruthi Kutty

Technical Editor

Jeeten Handu

Ankita Meshram

Project Coordinators

Wendell Palmer

Abhishek Kori

Proofreader

Maria Gould

Indexer

Priya Subramani

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

About the Author

David Herron has worked as a software engineer and software quality engineer in Silicon Valley for over 20 years. Recently he worked for Yahoo! as an architect of the Quality Engineering team for their new Node.js based web app platform (Manhattan and Mojito).

While a staff engineer at Sun Microsystems, David worked as an architect of the Java SE Quality Engineering team where he focused on test automation tools, including the AWT Robot class that's now widely used in GUI test automation software. He was involved with launching the OpenJDK project, and other open source activities related to Java.

Before Sun he worked for VxTreme on the video streaming stack which eventually became Windows Media Player when Microsoft bought that company. At The Wollongong Group he worked on both the e-mail client and server software and was part of several IETF working groups improving e-mail-related protocols.

David is interested in electric vehicles, world energy supplies, climate change, and environmental issues, and is a co-founder of *Transition Silicon Valley*. As an online journalist, he writes about electric cars and other green technology for PlugInCars.com, TorqueNews.com, Examiner.com, LongTailPipe.com, and ElectricRaceNews.com. He runs a large electric vehicle discussion website on VisForVoltage.org, and blogs about other topics including Node.js, Drupal, and Doctor Who on DavidHerron.com. Using Node.js, he has developed a Content Management System called AkashaCMS (akashacms.com) that produces static HTML websites.

Acknowledgement

I wish to thank my mother, Evelyn, for, well everything; my father, Jim; my sister, Patti; and my brother, Ken. What would life be without all of you?

I wish to thank my girlfriend, Maggie, for being there and encouraging me, her belief in me, her wisdom and humor, and kicks in the butt when needed. May we have many more years of this.

I wish to thank Dr. Ken Kubota of the University of Kentucky, for believing in me, and giving me my first job in computing. It was six years of learning, not just the art of computer system maintenance, but so much more.

I wish to thank my former employers, University of Kentucky Mathematical Sciences Department, The Wollongong Group, MainSoft, V Xtreme, Sun Microsystems, and Yahoo!, and all the people I worked with in each company.

I am grateful to Ryan Dahl, Isaac Schlueter, and the other Node core team members for having the wisdom and vision needed to create such a joy-filled and fluid software development platform. Some platforms are just plain hard to work with, but not this one, and that takes vision to implement it so well.

About the Reviewers

Ollie Bennett is a technical consultant based in London, with a passion for playing with the latest technologies. After completing a master's degree in Theoretical Physics, he focused his attention on web development, and maintains a portfolio of websites. Node.js and other recent JavaScript advancements are forming an ever increasing part of his interests.

He can be found at <http://olliebennett.co.uk/>.

Nathan Zondlo is a graduating senior from Lock Haven University. He has a strong interest in web development using the JavaScript full stack. He has built various websites using the HTML5/CSS3/JavaScript trifecta, but is currently spending all his free time researching JavaScript and contributing to various online communities.

I would love to thank Packt Publishing, namely Sruthi Kutty and Abhishek Kori, for reaching out to me through LinkedIn's Node.js group and asking me to take part in the review process. Troubleshooting and building the applications in the book has been a lot of fun, and I'm very happy to be included in the process.

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	1
Chapter 1: About Node	7
The capabilities of Node	8
Server-side JavaScript	9
Why should you use Node?	9
Threaded versus asynchronous event-driven architecture	11
Performance and utilization	13
Is Node a cancerous scalability disaster?	15
Server utilization, the bottom line, and green web hosting	16
What's in a name – Node, Node.js, or Node.JS?	17
Summary	17
Chapter 2: Setting up Node	19
System requirements	19
Installing Node using package managers	20
Installing on Mac OS X with MacPorts	20
Installing on Mac OS X with Homebrew	20
Installing on Linux from package management systems	21
Installing the Node distribution from nodejs.org	22
Installing Node on Windows using Chocolatey Gallery	23
Installing the StrongLoop Node distribution	23
Installing from source on POSIX-like systems	24
Installing prerequisites	24
Installing developer tools on Mac OS X	25
Installing from source for all POSIX-like systems	25
Maintaining multiple Node installs simultaneously	27
Run a few commands; testing the commands	28
Node's command-line tools	28

Running a simple script with Node	29
Launching a server with Node	30
npm – the Node package manager	31
Starting Node servers at system startup	33
Summary	36
Chapter 3: Node Modules	37
Defining a module	37
Node modules	39
Node's algorithm for resolving require(module)	39
Module identifiers and path names	39
Local modules within your application	40
Bundling external dependencies with your application	41
System-wide modules in NODE_PATH and elsewhere	43
Complex modules – modules as directories	44
Node package manager	45
The npm package format	45
Finding npm packages	47
Using the npm commands	48
Getting help with npm	49
Viewing package information	49
Installing an npm package	50
Installing native code modules on Windows	51
Installing packages local to a module	51
Eliminating duplicate modules installed beneath node_modules	52
Listing the currently installed packages	53
Package scripts	55
Editing and exploring installed package content	55
Updating outdated packages you've installed	56
Uninstalling an installed npm package	57
Developing and publishing npm packages	57
npm configuration settings	59
Package version strings and ranges	60
CommonJS modules	62
Demonstrating module encapsulation	63
Summary	64
Chapter 4: HTTP Servers and Clients – A Web Application's First Steps	65
Sending and receiving events with EventEmitter	65
EventEmitter theory	67
HTTP server applications	67
HTTP Sniffer – listening to the HTTP conversation	71
Web application frameworks	73

Getting started with Express	74
Walking through the default Express application	77
Calculating the Fibonacci sequence with Express	78
Computationally intensive code and the event loop	83
Algorithmic refactoring	85
Making HTTP Client requests	87
Calling a REST backend service from an Express application	89
Implementing a simple REST server with Express	90
Refactoring the Fibonacci application for REST	92
Some RESTful Node modules	93
Summary	94
Chapter 5: Implementing a Simple Express Application	95
Express and the MVC paradigm	95
Creating the Notes application code	96
The Notes model	96
The Notes home page	97
Adding a new note (create)	99
Viewing notes (read)	102
Editing an existing note (update)	104
Deleting notes (destroy)	105
Changing the look of an Express application	107
Scaling up and running multiple instances	109
Summary	111
Chapter 6: Data Storage and Retrieval	113
Asynchronizing the Notes application	114
Injecting the model configuration into routers	115
The notes router	116
Storing notes in files	119
Configuring app.js	122
Storing notes with the LevelUP data store	123
Installing LevelUP	123
LevelUP model code for Notes	124
Configuring app.js for LevelUP	125
Storing notes in SQL – SQLite3	126
Setting up a schema with SQLite3	126
Model code	127
Configuring app.js	129
Storing notes the ORM way with the Sequelize module	131
Schema setup and model code	131
Configuring app.js	134

Storing notes in MongoDB with Mongoose	135
Implementing the Notes model in Mongoose	136
Configuring app.js	139
Summary	140
Chapter 7: Multiuser Authorization, Deployment, Scaling, and Hosting	141
User authentication	141
Changes in app.js	142
The Sequelize-based users model	146
Routing module for the login, logout, and account pages	148
Initializing the user table	154
Running the Notes application	155
Deploying Notes on Debian	158
Scaling to use all cores on multi-core servers	161
Deploying Notes on cloud hosting (AppFog)	163
Summary	170
Chapter 8: Dynamic Interaction between the Client and Server Application	171
Adding real-time web features to Notes	172
Introducing Socket.IO	173
Initializing Socket.IO with Express	173
Setting up the client code	175
Events between the Notes server and client code	176
Modifying the Notes model to send events	177
Sending the events from the Notes server	179
Browser-side event handlers	180
Running the Notes application with Socket.IO	183
Listening to the heartbeat and cleaning up when it stops	185
Sending messages between users	187
Socket.IO events for sending messages between users	187
Data model to store messages	188
Setting up client-side code for sending messages	190
Dispatching messages between the client- and server-side	192
Displaying messages to the user	193
Running Notes and sending messages	196
Summary	197

Chapter 9: Unit Testing	199
Testing asynchronous code	199
Assert – the simplest testing methodology	201
Testing a model	202
Executing the tests	207
Testing router functions	208
Diagnosing a failing test case	213
Making it easy to run the tests	217
Summary	218
Index	221

Preface

Welcome to the world of developing web software with Node (also known as Node.js). Node is an up-and-coming software platform that liberates JavaScript from the web browser, enabling it to be used as a general software development platform in server-side applications. It runs atop the ultra-fast JavaScript engine from the Chrome browser, V8, and adds in a fast and robust library of asynchronous network I/O modules. The primary focus of Node is on building high performance, highly scalable, server and client applications for the "real-time Web".

The platform was developed by Ryan Dahl in 2009 after a couple of years of experimenting with web server component development in Ruby and other languages. Today, Node's development is sponsored by Joyent, and is led by a core team of several members from several companies. Dahl's exploration led him to the architectural choice of using asynchronous event-driven systems rather than the traditional thread-based concurrency model.

This model was chosen because it's simpler (threaded systems are notoriously difficult to develop), has lower overhead over maintaining a thread per connection, and for speed. The goal of Node is to provide an "easy way to build scalable network servers". The design is similar to, and influenced by, other systems, such as Event Machine (Ruby) and the Twisted framework (Python).

JavaScript was chosen because the anonymous functions and other language elements are an excellent method for implementing asynchronous computation, such as event handlers. Handler functions are often able to be written in-line with the invoking code, just by providing an anonymous function. The runtime library of built-in modules are ingeniously designed to make good use of asynchronous coding.

The result is a platform allowing developers to not only succinctly write code of great power, but to have a lot of fun while doing so.

JavaScript on both the server and client (browser) lets us implement a vision dating back to the days when Java's proponents first showed us dynamic stuff running inside a web page. Java never became the language for this, and JavaScript is quickly advancing to take on this vision in a real big way. By having JavaScript on both ends of the wire, we can use common code, common data structures, and in general have an easier time of things.

This book, *Node Web Development, Second Edition*, focuses on building web applications using Node. We assume you have some knowledge of JavaScript, and maybe even have server-side development experience. We will be taking a tour through the important concepts to understand programming with Node.

To do so, we'll be developing several applications, including one that's somewhat useful, and this will be developed in stages over several chapters. That application will, in the end, support real-time updates to multiple users simultaneously, as well as real-time messaging between users.

We'll be dissecting the code to scrutinize how it works, and discussing how to apply the ideas to your own programs. We'll install Node and npm, and learn how to install or develop npm packages and Node modules. We'll look at how to distribute heavy workloads to backend servers, implement simple REST services, and even how to do real-time communication between the server and client.

What this book covers

Chapter 1, About Node, introduces you to the Node platform. It covers its uses, the technological architectural choices in Node, its history, the history of server-side JavaScript, and why JavaScript should be liberated from the browser.

Chapter 2, Setting up Node, goes over setting up a Node developer environment, including compiling and installing from source code. We briefly touch on Node deployment to production servers.

Chapter 3, Node Modules, explores the module as the unit of modularity in developing Node applications. We dive deep into understanding and developing Node modules, at npm, the Node Package Manager, and several scenarios using npm to manage installed packages, or developing npm packages for distribution to others.

Chapter 4, HTTP Servers and Clients – A Web Application's First Steps, starts exploring web development with Node, with the fundamentals of Node now in hand. We develop several small web-server and web-client applications in Node. We use the Fibonacci algorithm to explore the effects of heavy-weight long-running computations on a Node web application, and several mitigation techniques, including developing a simple REST-based backend server to compute Fibonacci numbers.

Chapter 5, Implementing a Simple Express Application, begins several chapters of developing a somewhat useful application with which we'll be exploring the Node platform. In this chapter we build the core skeleton of the application.

Chapter 6, Data Storage and Retrieval, helps add data storage capabilities to the application for data persistence, application robustness, and sharing data between application instances. The data storage engines we look at include the file system, LevelUP, SQLite3, Sequelize, MySQL, and MongoDB (using Mongoose).

Chapter 7, Multiuser Authorization, Deployment, Scaling, and Hosting, shows how to add user authentication capabilities to the application, allowing it to be used by both logged-in and anonymous users. We then look at several ways of deploying the application on real servers, including a cloud hosting platform.

Chapter 8, Dynamic Interaction between the Client and Server Application, explores the real-time web by using the Socket.IO library to enable real-time communication between the server and client. We experience firsthand the power of having JavaScript on both server and client.

Chapter 9, Unit Testing, shows how to implement unit testing and test first development methodologies with a Node application. We use this to find a few bugs in the code written for the previous chapters, and then walk through fixing those bugs.

What you need for this book

The basic requirement is to have Node installed, and to have a programmer-oriented text editor. We show you how to install Node either from a prebuilt binary package, or from source. The most important tool is the one between your ears. The examples have been tested against Node v0.10.x.

Some of the chapters require database engines such as MySQL and MongoDB.

Some of the modules that we'll install require compiling from source code. That requires a C/C++ compiler toolchain plus Python 2.7 or later (but not Python 3.0 or later).

While this book is about developing web applications, it does not require you to have a web server. Node provides its own web server stack.

Who this book is for

This book is written for any software engineer who wants the adventure that comes with a new software platform embodying a new programming paradigm.

Server-side engineers may find the concepts behind Node refreshing, giving you a different perspective on web application development. JavaScript is a powerful language and Node's asynchronous nature plays to JavaScript's strengths.

Developers experienced with JavaScript in the browser may find it fun to bring that knowledge to new territory.

We assume that you already know how to write software, and have an understanding of modern programming languages such as JavaScript.

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 are shown as follows: "With the `node_modules` directory situated where it is, any module within `drawapp` can access `express` with the code."

A block of code is set as follows:

```
#!/usr/bin/env node
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337);
```

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


```
exports.view = function(req, res, next) {
  var user = req.user ? req.user : undefined;
  if (req.query.key) {


    readNote(req.query.key, user, res, function(err, data)
```

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

```
$ sudo forever list
info: Forever processes running
data: uid command script forever pid logfile uptime
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Once you're registered, go to the dashboard and click on the **Create App** button".

[ 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.

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.

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

About Node

Node is an exciting new platform for developing web applications, application servers, any sort of network server or client, and general purpose programming. It is designed for extreme scalability in networked applications through an ingenious combination of server-side JavaScript, asynchronous I/O, and asynchronous programming, and is built around JavaScript anonymous functions, and a single execution thread event-driven architecture.

The Node model is very different from common application server platforms that scale using threads. The claim is that, because of the event-driven architecture, the memory footprint is low, the throughput is high, the latency profile under load is better, and the programming model is simpler. The Node platform is in a phase of rapid growth, and many are seeing it as a compelling alternative to the traditional – Apache, Java, PHP, Python, Ruby on Rails, and so on – approach to building web applications.

At heart it is a standalone JavaScript virtual machine, with extensions making it suitable for general purpose programming, and with a clear focus on application server development. The Node platform isn't directly comparable to programming languages frequently used for developing web applications, and neither is it directly comparable to the containers which deliver the HTTP protocol to web clients (Apache httpd, Tomcat, GlassFish, and so on). At the same time, many regard it as potentially supplanting the traditional web application development stacks.

It is implemented around a non-blocking I/O event loop and a layer of file and network I/O libraries, all built on top of the V8 JavaScript engine (from the Chrome web browser). The I/O library is enough to implement any sort of server implementing any TCP or UDP protocol, such as, DNS, HTTP, IRC, or FTP. While it supports developing servers or clients for any network protocol, the biggest use case is in regular websites in place of a technology such as an Apache/PHP or Rails stack, or to complement existing websites. For example, adding real-time chat or monitoring to existing websites can be easily done with the Socket.IO library for Node.

This book will give you an introduction to Node. We presume that you already know how to write software, are familiar with JavaScript, and know something about developing web applications in other languages. We will dive right into developing working applications and recognize that often the best way to learn is by rummaging around in working code.

The capabilities of Node

Node is a platform for writing JavaScript applications outside web browsers. This is not the JavaScript we are familiar with in web browsers! There is no DOM built into Node, nor any other browser capability, but a DOM can be added using JSDom, and there are Node-based wrappers available for some browser engines. Between the JavaScript language and its asynchronous I/O framework, it is a powerful application development platform.

Beyond its native ability to execute JavaScript, the bundled modules provide the following capabilities:

- Command-line tools (in shell script style)
- Interactive-TTY style of program (REPL or Read-Eval-Print Loop)
- Excellent process control functions to oversee child processes
- A buffer object to deal with binary data
- TCP or UDP sockets with comprehensive event-driven callbacks
- DNS lookup
- Layered on top of the TCP library is an HTTP and HTTPS client/server
- File system access
- Built-in rudimentary unit testing support through assertions

The network layer of Node is low level while being simple to use. For example, the HTTP modules allow you to write an HTTP server (or client) in a few lines of code. This is powerful, but it puts you, the programmer, very close to the protocol requests and makes you implement precisely which HTTP headers to return in request responses. Where a PHP programmer generally doesn't care about the headers, a Node programmer does.

In other words, it's very easy to write an HTTP server in Node, but the typical web application developer doesn't need to work at that level of detail. For example, PHP coders assume Apache is already there, and that they don't have to implement the HTTP server portion of the stack. The Node community has developed a wide range of web application frameworks, such as Connect, allowing developers to quickly configure an HTTP server that provides all of the basics we've come to expect — sessions, cookies, serving static files, logging, and so on — thus letting developers focus on their business logic.

Server-side JavaScript

Quit scratching your head already. Of course you're doing it, scratching your head and mumbling to yourself, "What's a browser language doing on the server?" In truth, JavaScript has a long and largely unknown history outside the browser. JavaScript is a programming language, just like any other language, and the better question to ask is "Why should JavaScript remain trapped inside browsers?"

Back in the dawn of the web age, the tools for writing web applications were at a fledgling stage. Some were experimenting with Perl or TCL to write CGI scripts, the PHP and Java languages had just been developed, and even JavaScript was being used on the server side. One early web application server was Netscape's LiveWire server, which used JavaScript. Some versions of Microsoft's ASP used JScript, their version of JavaScript. A more recent server-side JavaScript project is the RingoJS application framework in the Java universe. It's built on top of Rhino, a JavaScript implementation written in Java. All this means that JavaScript outside the browser is not a new thing, even if it is uncommon.

Node brings to the table a combination never seen before; namely, the coupling of fast event-driven I/O and a fast modern JavaScript engine such as Google's V8, the ultrafast JavaScript engine at the heart of Google's Chrome web browser.

Why should you use Node?

The JavaScript language is very popular due to its ubiquity in web browsers. It compares favorably against other languages while having many modern advanced language concepts. Thanks to its popularity, there is a deep talent pool of experienced JavaScript programmers out there.

JavaScript is a dynamic programming language with loosely typed and dynamically extendable objects, that can be informally declared as needed. Functions are a first class object routinely used as anonymous closures (nameless functions that can be passed around with ease). This makes JavaScript more powerful than some other languages commonly used for web applications. In theory these features make developers more productive.

There is a raging debate between dynamic and non-dynamic languages, or rather between statically typed and loosely typed ones. Loosely typed dynamic languages such as JavaScript are thought to give programmers more freedom to quickly write code. Proponents of strongly typed languages, such as Java, argue that the compiler helps to catch programming mistakes that are not caught in loosely typed languages. The debate is not settled, and may never be settled. The Node platform, by using JavaScript, of course sits in the loosely typed languages camp.

One of the main disadvantages of JavaScript is the global object. In a web page, all the top-level variables are tossed together in the global object, which can create an unruly chaos when mixing modules together. Since web applications tend to have lots of objects, probably coded by multiple organizations, one may think programming in Node would be a minefield of conflicting global objects. Instead, Node uses the CommonJS module system, meaning that variables local to a module are truly local to the module, even if they look like global variables. This clean separation between modules negates the global object problem.

Having the same programming language on the server and client has been a long-time dream on the web. This dream dates back to the early days of Java, where applets were to be the front end to server applications written in Java, and JavaScript was originally envisioned as a lightweight scripting language for applets. Something fell down along the way, and we ended up with JavaScript as the principle in-browser client-side language, rather than Java. With Node we may finally be able to implement applications with the same programming language on the client and server, by having JavaScript at both ends of the Web, in the browser and server.

A common language for the frontend and backend offers several potential wins:

- The same programming staff can work on both ends of the wire
- Code can be migrated between server and client more easily
- Common data formats (JSON) between server and client
- Common software tools for server and client
- Common testing or quality reporting tools for server and client

- When writing web applications, view templates can be used on both sides
- A similar language between server and client teams could make for better communication among team members

Node facilitates implementing all these positive benefits (and more) with a compelling platform and development community.

Threaded versus asynchronous event-driven architecture

The asynchronous event-driven architecture of Node is said to be the cause of its blistering performance. Well, that and Chrome's V8 JavaScript engine. The normal application server model uses blocking I/O to retrieve data, and uses threads for concurrency. Blocking I/O causes threads to wait, causing churn between threads as they are forced to wait on I/O while the application server handles requests. Threads add complexity to the application server, as well as server overhead.

Node has a single execution thread with no waiting on I/O or context switching. Instead there is an event loop, looking for events and dispatching them to handler functions. The paradigm is to pass an anonymous function into any operation that will take time to complete. The handler function is invoked when the operation is complete, and in the meantime the event loop continues dispatching events.

This model is typical in GUI applications, as well as for JavaScript execution in a web browser. Like with those systems, event handler functions must quickly return to the event loop for dispatching the next immediately runnable task.

To help us wrap our heads around this, Ryan Dahl, the creator of Node, (in his "Cinco de Node" presentation) asked us what happens while executing a code like this:

```
result = query('SELECT * from db');  
// operate on the result
```

Of course, the program pauses at that point while the database layer sends the query to the database, which determines the result, and returns the data. Depending on the query that pause can be quite long. Well, a few milliseconds, but that is an eon in computer time. This pause is bad because while the entire thread is idling, another request might come in, and for thread-based server architectures that means a thread context switch. The more outstanding connections to the server, the greater the number of thread context switches. Context switching is not free, because more threads requires more memory for per-thread state and more time for the CPU to spend on thread management overhead.

Simply by using asynchronous, event-driven I/O, Node removes most of this overhead while introducing very little of its own.

Frequently implementing concurrency with threads comes with admonitions like these: "expensive and error-prone", "the error-prone synchronization primitives of Java", or "designing concurrent software can be complex and error prone". The complexity comes from the access to shared variables and various strategies to avoid deadlock and competition between threads. The "synchronization primitives of Java" are an example of such a strategy, and obviously many programmers find them hard to use. There's the tendency to create frameworks such as `java.util.concurrent` to tame the complexity of threaded concurrency, but some might argue that papering over complexity does not make things simpler.

Node asks us to think differently about concurrency. Callbacks fired asynchronously from an event loop are a much simpler concurrency model; simpler to understand, and simpler to implement.

Ryan Dahl points to the relative access time of objects to understand the need for asynchronous I/O. Objects in memory are more quickly accessed (on the order of nanoseconds) than objects on disk or objects retrieved over the network (milliseconds or seconds). The longer access time for external objects is measured in zillions of clock cycles, which can be an eternity when your customer is sitting at their web browser ready to be bored and move on if it takes longer than two seconds to load the page.

In Node, the query discussed previously would read as follows:

```
query('SELECT * from db', function (err, result) {  
  if (err) throw err; // handle errors  
  // operate on result  
});
```

This code performs the same query written earlier. The difference is that the query result is not the result of the function call, but is provided to a callback function that will be called later. The order of execution is not one line after another, but instead determined by the order of callback function execution.

In this example, the `query` function will return almost immediately to the event loop, which goes on to service other requests. One of those requests will be the response to the query, which invokes the callback function. Quickly returning to the event loop ensures higher server utilization. That's great for the owner of the server, but there's an even bigger gain which might help the user to experience quicker page content construction.

Commonly, web pages bring together data from dozens of sources. Each one has a query and response as discussed earlier. By using asynchronous queries each one can happen in parallel, where the page construction function can fire off dozens of queries — no waiting, each with their own callback — then go back to the event loop, invoking the callbacks as each is done. Because it's in parallel, the data can be collected much more quickly than if these queries were done synchronously, one at a time. Now the reader on their web browser is happier because the page loads more quickly.

Performance and utilization

Some of the excitement over Node is due to its throughput (requests per second it can serve). Comparative benchmarks of similar applications, for example, Apache and Node, show that Node has tremendous performance gains.

One benchmark going around is this simple HTTP server (borrowed from nodejs.org), which simply returns a "Hello World" message, directly from memory:

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

This is one of the simpler web servers one can build with Node. The `http` object encapsulates the HTTP protocol, and its `http.createServer` method creates a whole web server, listening on the port specified in the `listen` method. Every request (whether a GET or POST on any URL) on that web server calls the provided function. It is very simple and lightweight. In this case, regardless of the URL, it returns a simple text/plain Hello World response.

Because of its minimal nature, this simple application should demonstrate the maximum request throughput of Node. Indeed many have published benchmark studies starting from this simplest of HTTP servers.

Ryan Dahl (Node's original author) showed a simple benchmark (http://nodejs.org/cinco_de_node.pdf) which returned a 1 megabyte binary buffer; Node gave 822 req/sec while Nginx gave 708 req/sec, for a 15 percent improvement over Nginx. He also noted that Nginx peaked at 4 megabytes memory, while Node peaked at 64 megabytes.

Yahoo! search engineer Fabian Frank published a performance case study of a real-world search query suggestion widget implemented with Apache/PHP and two variants of Node stacks (<http://www.slideshare.net/FabianFrankDe/nodejs-performance-case-study>). The application is a pop-up panel showing search suggestions as the user types in phrases, using a JSON-based HTTP query. The Node version could handle eight times the number of requests/second with the same request latency. Fabian Frank said both Node stacks scaled linearly until CPU usage hit 100 percent. In another presentation (<http://www.slideshare.net/FabianFrankDe/yahoo-scale-nodejs>), he discussed how Yahoo! Axis is running on Manhattan + Mojito and the value of being able to use the same language (JavaScript) and framework (YUI/YQL) on both the frontend and backend.

LinkedIn did a massive overhaul of their mobile app, using Node for the server-side to replace an old Ruby on Rails app. The switch let them move from 30 servers down to three, and to merge the frontend and backend team because everything was written in JavaScript. Before choosing Node, they'd evaluated Rails with Event Machine, Python with Twisted, and Node, choosing Node for the reasons just given (<http://arstechnica.com/information-technology/2012/10/a-behind-the-scenes-look-at-linkedins-mobile-engineering/>).

Mikito Takada blogged about benchmarking and performance improvements in a "48 hour hackathon" application he built (<http://blog.mixu.net/2011/01/17/performance-benchmarking-the-node-js-backend-of-our-48h-product-wehearvoices-net/>) comparing Node with what he claims is a similar application written with Django (a web application framework for Python). The unoptimized Node version is quite a bit slower (response time) than the Django version but a few optimizations (MySQL connection pooling, caching, and so on) made drastic performance improvements, handily beating out Django.

A key realization about Node performance is the need to quickly return to the event loop. We go over this in *Chapter 4, HTTP Servers and Clients – A Web Application's First Steps*, in more detail, but if a callback handler takes too long to execute, it will prevent Node from being the blistering fast server it was designed to be. In one of Ryan Dahl's earliest blog posts about the Node project he discussed a requirement that event handlers execute within 5 ms. Most of the ideas in that post were never implemented, but Alex Payne wrote an intriguing blog post on this (<http://a13x.net/2010/07/27/node.html>), drawing a distinction between "scaling in the small" and "scaling in the large".

Small-scale web applications (**scaling in the small**) should have performance and implementation advantages when written for Node, instead of the "P" languages (Perl, PHP, Python, and so on) normally used. JavaScript is a powerful language, and the Node environment with its modern fast virtual machine design offers performance and concurrency advantages over interpreted languages such as PHP.

He goes on to argue that "**scaling in the large**", enterprise-level applications, will always be hard and complex. One typically throws in load balancers, caching servers, multiple redundant machines, in geographically dispersed locations, to serve millions of users from around the world with a fast web browsing experience. Perhaps the application development platform isn't as important as the whole system.

Is Node a cancerous scalability disaster?

In October 2011, software developer and blogger Ted Dziuba wrote an infamous blog post (since pulled from his blog) claiming that Node is a cancer, calling it a "scalability disaster." The example he showed as proof is a CPU-bound implementation of the Fibonacci sequence algorithm. While his argument was flawed, he raised a valid point that Node application developers have to consider. Where do you put the heavy computational tasks?

The Fibonacci sequence, serving as a stand-in for heavy computational tasks, quickly becomes computationally expensive to calculate, especially for a naïve implementation. The previous version of this book used an identical Fibonacci implementation, and was used to demonstrate why event handlers have to return quickly to the event loop.

```
var fibonacci = exports.fibonacci = function(n) {  
  if (n === 1 || n === 2)  
    return 1;  
  else  
    return fibonacci(n-1) + fibonacci(n-2);  
}
```

Yes, there are many ways to calculate Fibonacci numbers more quickly. We are showing this as a general example of what happens to Node when event handlers are slow, and not to debate the best ways to calculate mathematics functions.

If you call this from the request handler in a Node HTTP server, for sufficiently large values of *n* (say, 40) the server becomes completely unresponsive because the event loop is not running, because this function is grinding through the calculation.

Does this mean Node is a flawed platform? No. It just means that the programmer must take care to identify code with long-running computations, and develop a solution. The possible solutions include rewriting the algorithm to work with the event loop, or to foist computationally expensive calculations to a backend server.

Additionally, there is nothing CPU intensive about retrieving data items from a database, plugging it into a template to send through the HTTP response. That's what typical web applications do after all.

With the Fibonacci algorithm, a simple rewrite dispatches the computations through the event loop, letting the server continue handling requests on the event loop. By using callbacks and closures (anonymous functions) we're able to maintain asynchronous I/O and concurrency promises.

```
var fibonacciAsync = exports.fibonacciAsync = function(n, done) {
  if (n === 1 || n === 2) done(1);
  else {
    process.nextTick(function() {
      fibonacciAsync(n-1, function(val1) {
        process.nextTick(function() {
          fibonacciAsync(n-2, function(val2) {
            done(val1+val2);
          });
        });
      });
    });
  }
}
```

Dziuba's valid point wasn't expressed well in his blog post, and was somewhat lost in the flames following that post. His point was that while Node is a great platform for I/O-bound applications, it isn't a good platform for computationally intensive ones.



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.

Server utilization, the bottom line, and green web hosting

The striving for optimal efficiency (handling more requests/second) is not just about the geeky satisfaction that comes from optimization. There are real business and environmental benefits. Handling more requests per second, as Node servers can do, means the difference between buying lots of servers and buying only a few servers. Node can let your organization do more with less.

Roughly speaking, the more servers one buys, the greater the cost, and the greater the environmental impact. There's a whole field of expertise around reducing cost and environmental impact of running web server facilities, to which that rough guideline doesn't do justice. The goal is fairly obvious; fewer servers, lower costs, and lower environmental impact.

Intel's paper, *Increasing Data Center Efficiency with Server Power Measurements* (http://download.intelintel.com/it/pdf/Server_Power_Measurement_final.pdf), gives an objective framework for understanding efficiency and data center costs. There are many factors such as building, cooling system, and computer system design. Efficient building design, efficient cooling systems, and efficient computer systems (datacenter efficiency, datacenter density, and storage density) can decrease costs and environmental impact. But you can destroy those gains by deploying an inefficient software stack compelling you to buy more servers than if you had an efficient software stack. Alternatively you can amplify gains from datacenter efficiency with an efficient software stack.

This talk about efficient software stacks isn't just for altruistic environmental purposes. This is one of those cases where being green can help your business bottom line.

What's in a name – Node, Node.js, or Node.JS?

The name of the platform is Node.js, but throughout this book we are spelling it as Node because we are following a cue from the `nodejs.org` website, which says the trademark is Node.js (lower case .js) but throughout the site they spell it as Node. We are doing the same in this book.

Summary

We've learned a lot in this chapter, specifically:

- That JavaScript has a life outside web browsers
- The difference between asynchronous and blocking I/O
- The attributes of Node, and where it fits in the overall web application platform market
- Threaded versus asynchronous software
- The advantages of fast event-driven asynchronous I/O, coupled with a language with great support for anonymous closures
- Node performance

Now that we've had this introduction to Node, we're ready to dive in and start using it. In *Chapter 2, Setting up Node*, we'll go over setting up a Node environment, so let's get started.

2

Setting up Node

Before getting started with using Node you must set up your development environment. In the following chapters we'll be using this for development, and for non-production deployment.

In this chapter we will:

- See how to install Node from source and prepackaged binaries on Linux, Mac, or Windows
- See how to install the npm package manager, and some popular tools
- Learn a bit about the Node module system

So let's get on with it.

System requirements

Node runs on POSIX-like operating systems, the various UNIX derivatives (Solaris, and so on), or workalikes (Linux, Mac OS X, and so on), as well as on Microsoft Windows, thanks to the extensive assistance from Microsoft. Indeed, many of the Node built-in functions are direct corollaries to POSIX system calls. It can run on machines both large and small, including the tiny ARM devices such as the Raspberry Pi microscale embeddable computer for DIY software/hardware projects.

Node is now available via package management systems, limiting the need to compile and install from source.

Installing from source requires having a C compiler (such as GCC), and Python 2.7 (or later). If you plan to use encryption in your networking code you will also need the OpenSSL cryptographic library. The modern UNIX derivatives almost certainly come with these, and Node's configure script (see later when we download and configure the source) will detect their presence. If you should have to install them, Python is available at <http://python.org> and OpenSSL is available at <http://openssl.org>.

Installing Node using package managers

The preferred method for installing Node, now, is to use the versions available in package managers such as apt-get, or MacPorts. Package managers simplify your life by helping to maintain the current version of the software on your computer and ensuring to update dependent packages as necessary, all by typing a simple command such as `apt-get update`. Let's go over this first.

Installing on Mac OS X with MacPorts

The MacPorts project (<http://www.macports.org/>) has for years been packaging a long list of open source software packages for Mac OS X, and they have packaged Node. After you have installed MacPorts using the installer on their website, installing Node is pretty much this simple:

```
$ sudo port search nodejs
nodejs @0.10.6 (devel, net)
    Evented I/O for V8 JavaScript

nodejs-devel @0.11.2 (devel, net)
    Evented I/O for V8 JavaScript

Found 2 ports.
--
npm @1.2.21 (devel)
    node package manager
$ sudo port install nodejs npm
.. long log of downloading and installing prerequisites and Node
```

Installing on Mac OS X with Homebrew

Homebrew is another open source software package manager for Mac OS X, which some say is the perfect replacement for MacPorts. It is available through their home page at <http://mxcl.github.com/homebrew/>. After installing Homebrew using the instructions on their website, using it to install Node is as simple as this:

```
$ brew search node
leafnode  node
$ brew install node
==> Downloading http://nodejs.org/dist/v0.10.7/node-v0.10.7.tar.gz
```

```
#####
100.0%
==> ./configure --prefix=/usr/local/Cellar/node/0.10.7
==> make install
==> Caveats
Homebrew installed npm.
We recommend prepending the following path to your PATH environment
variable to have npm-installed binaries picked up:
    /usr/local/share/npm/bin
==> Summary
/usr/local/Cellar/node/0.10.7: 870 files, 16M, built in 21.9 minutes
```

Installing on Linux from package management systems

While it's still premature for Linux distributions or other operating systems to prepackage Node with their OS, that doesn't mean you cannot install it using the package managers. Instructions on the Node wiki currently list packaged versions of Node for Debian, Ubuntu, OpenSUSE, and Arch Linux.

See: <https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager>

For example, on Debian sid (unstable):

```
# apt-get update
# apt-get install nodejs # Documentation is great.
```

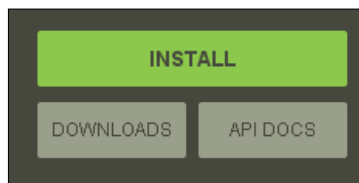
And on Ubuntu:

```
# sudo apt-get install python-software-properties
# sudo add-apt-repository ppa:chris-lea/node.js
# sudo apt-get update
# sudo apt-get install nodejs npm
```

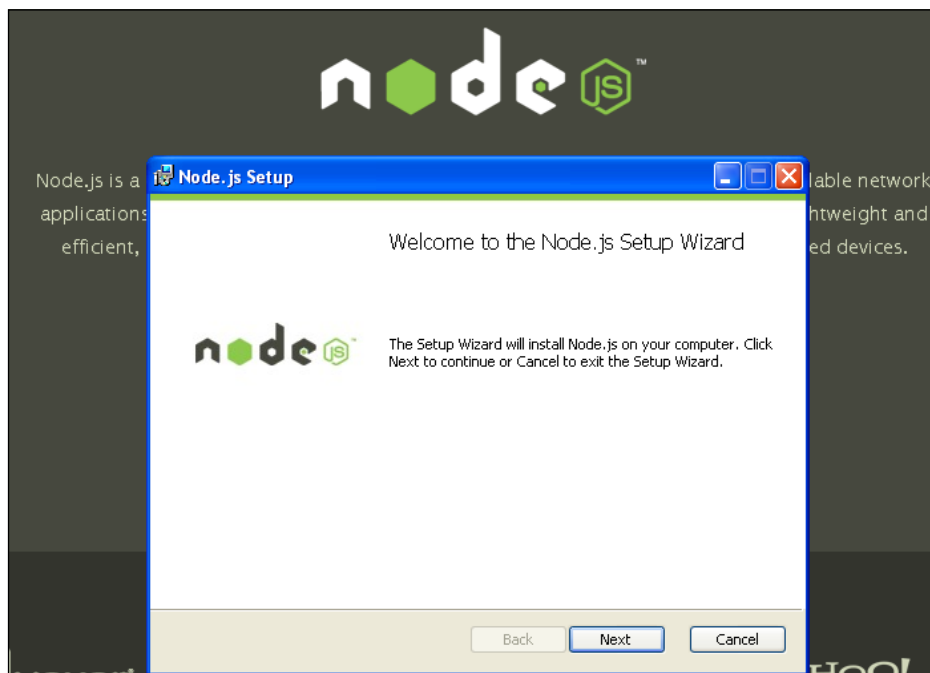
We can expect in due course that the Linux distros and other operating systems will routinely bundle Node into the OS like they do with other languages today.

Installing the Node distribution from nodejs.org

The `nodejs.org` website offers prebuilt binaries for Windows, Mac OS X, Linux, and Solaris. You simply go to the website, click on the **Install** button, and run the installer. For systems with package managers, such as the ones we've just discussed, it's preferable to use that installation method. That's because you'll find it easier to stay up-to-date with the latest version. However, on Windows this method may be preferred.



For Mac OS X, the installer is a PKG file giving the typical installation process. For Windows, the installer simply takes you through the typical install wizard process.



Once finished with the installer, you have a command line tool with which to run Node programs.

The pre-packaged installers are the simplest ways to install Node, for those systems for which they're available.

Installing Node on Windows using Chocolatey Gallery

Chocolatey Gallery is a package management system, built on top of NuGet. Using it requires a Windows machine modern enough to support the Powershell and the .NET Framework 4.0. Once you have Chocolatey Gallery (<http://chocolatey.org/>), installing Node is as simple as this:

```
C:\> cinst install nodejs
```

Installing the StrongLoop Node distribution

StrongLoop (<http://strongloop.com>) has put together a supported version of Node that is prepackaged with several useful tools. This is a Node distribution in the same sense in which Fedora or Ubuntu are Linux distributions. StrongLoop brings together several useful packages, some of which were written by StrongLoop. StrongLoop tests the packages together, and distributes installable bundles through their website.

The packages in the distribution include Express, Passport, Mongoose, Socket.IO, Engine.IO, Async, and Request. We will use all of those modules in this book.

To install, navigate to the company home page and click on the **Products** link. They offer downloads of precompiled packages for both RPM and Debian Linux systems, as well as Mac OS X and Windows. Simply download the appropriate bundle for your system.

For the RPM bundle, type the following:

```
$ sudo rpm -i bundle-file-name
```

For the Debian bundle, type the following:

```
$ sudo dpkg -i bundle-file-name
```

The Windows or Mac bundles are the usual sort of installable packages for each system. Simply double-click on the installer bundle, and follow the instructions in the install wizard.

Once StrongLoop Node is installed, it provides not only the `node` and `npm` commands (we'll go over these in a few pages), but also the `slnode` command. That command offers a superset of the `npm` commands, such as boilerplate code for modules, web applications, or command-line applications.

Installing from source on POSIX-like systems

Installing the pre-packaged Node distributions is currently the preferred installation method. However, installing Node from source is desirable in a few situations:

- It could let you optimize the compiler settings as desired
- It could let you cross-compile, say for an embedded ARM system
- You might need to keep multiple Node builds for testing
- You might be working on Node itself

Now that you have the high-level view, let's get our hands dirty mucking around in some build scripts. The general process follows the usual `configure`, `make`, and `make install` routine that you may already have performed with other open source software packages. If not, don't worry, we'll guide you through the process.

The official installation instructions are in the Node wiki at <https://github.com/joyent/node/wiki/Installation>.

Installing prerequisites

As noted a minute ago, there are three prerequisites, a C compiler, Python, and the OpenSSL libraries. The Node installation process checks for their presence and will fail if the C compiler or Python is not present. The specific method of installing these is dependent on your operating system.

These commands will check for their presence:

```
$ cc --version
i686-apple-darwin10-gcc-4.2.1 (GCC) 4.2.1 (Apple Inc. build 5666) (dot 3)
Copyright (C) 2007 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
```

```
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR  
PURPOSE.
```

```
$ python
```

```
Python 2.6.6 (r266:84292, Feb 15 2011, 01:35:25)
```

```
[GCC 4.2.1 (Apple Inc. build 5664)] on darwin
```

```
Type "help", "copyright", "credits" or "license" for more  
information.
```

```
>>>
```

Installing developer tools on Mac OS X

The developer tools (such as GCC) are an optional installation on Mac OS X. There are two ways to get those tools, both of which are free. On the OS X installation DVD is a directory labeled `Optional Installs`, in which there is a package installer for —among other things— the developer tools, including Xcode.

The other method is to download the latest copy of Xcode (for free) from <http://developer.apple.com/xcode/>.

Most other POSIX-like systems, such as Linux, include a C compiler with the base system.

Installing from source for all POSIX-like systems

First, download the source from <http://nodejs.org/download>. One way to do this is with your browser, and another way is as follows:

```
$ mkdir src
```

```
$ cd src
```

```
$ wget http://nodejs.org/dist/v0.10.7/node-v0.10.7.tar.gz
```

```
$ tar xvfz node-v0.10.7.tar.gz
```

```
$ cd node-v0.10.7
```

The next step is to configure the source so that it can be built. It is done with the typical sort of configure script and you can see its long list of options by running the following:

```
$ ./configure --help.
```


To cause the installation to land in your home directory, run it this way:

```
$ ./configure --prefix=$HOME/node/0.10.7
..output from configure
```

If you want to install Node in a system-wide directory simply leave off the `--prefix` option, and it will default to installing in `/usr/local`.

After a moment it'll stop and more likely configure the source tree for installation in your chosen directory. If this doesn't succeed it will print a message about something that needs to be fixed. Once the configure script is satisfied, you can go on to the next step.

With the configure script satisfied, compile the software:

```
$ make
.. a long log of compiler output is printed
$ make install
```

If you are installing into a system-wide directory do the last step this way instead:

```
$ make
$ sudo make install
```

Once installed you should make sure to add the installation directory to your `PATH` variable as follows:

```
$ echo 'export PATH=$HOME/node/0.10.7/bin:${PATH}' >> ~/.bashrc
$ . ~/.bashrc
```

For `cs`h users, use this syntax to make an exported environment variable:

```
$ echo 'setenv PATH $HOME/node/0.10.7/bin:${PATH}' >> ~/.cshrc
$ source ~/.cshrc
```

This should result in some directories like this:

```
$ ls ~/node/0.10.7/
bin  include  lib  share
$ ls ~/node/0.10.7/bin
node    node-waf  npm
```

Maintaining multiple Node installs simultaneously

Normally you won't have multiple versions of Node installed, and doing so adds complexity to your system. But if you are hacking on Node itself, or are testing against different Node releases, or any of several similar situations, you may want to have multiple Node installations. The method to do so is a simple variation on what we've already discussed.

If you noticed during the instructions discussed earlier, the `-prefix` option was used in a way that directly supports installing several Node versions side-by-side in the same directory:

```
$ ./configure -prefix=$HOME/node/0.10.7
```

And:

```
$ ./configure -prefix=/usr/local/node/0.10.7
```

This initial step determines the install directory. Clearly when Version 0.10.7, Version 0.12.15, or whichever version is released, you can change the install prefix to have the new version installed side-by-side with the previous versions.

To switch between Node versions is simply a matter of changing the `PATH` variable (on POSIX systems), as follows:

```
$ export PATH=/usr/local/node/0.10.7/bin:${PATH}
```

It starts to be a little tedious to maintain this after a while. For each release, you have to set up Node, `npm`, and any third-party modules you desire in your Node install; also the command shown to change your `PATH` is not quite optimal. Inventive programmers have created several version managers to make this easier by automatically setting up not only Node, but `npm` also, and providing commands to change your `PATH` the smart way:

- Node version manager: <https://github.com/visionmedia/n>
- Nodefront, aids in rapid frontend development: <http://karthikv.github.io/nodefront/>

Run a few commands; testing the commands

Now that you've installed Node, we want to do two things; verify that the installation was successful, and familiarize you with the command-line tools.

Node's command-line tools

The basic install of Node includes two commands, `node` and `node-waf`. We've already seen `node` in action. It's used either for running command-line scripts or server processes. The other, `node-waf`, is a build tool for Node native extensions. Since it's for building native extensions, we will not cover it in this book and you should consult the online documentation at nodejs.org.

The easiest way to verify whether your Node installation works is also the best way to get help with Node. Type the following:

```
$ node --help
```

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

Options:

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

Environment variables:

<code>NODE_PATH</code>	<code>':'</code> -separated list of directories prefixed to the module search path.
<code>NODE_MODULE_CONTEXTS</code>	Set to 1 to load modules in their own global contexts.
<code>NODE_DISABLE_COLORS</code>	Set to 1 to disable colors in the REPL

Documentation can be found at <http://nodejs.org/>.

It prints the `USAGE` message, giving you the command-line options.

Notice that there are options for both Node and V8 (not shown in the previous command line). Remember that Node is built on top of V8; it has its own universe of options that largely focus on details of bytecode compilation or the garbage collection and heap algorithms. Enter `node --v8-options` to see the full list of them.

On the command line you can specify options, a single script file, and a list of arguments to that script. We'll discuss script arguments further in the next section, *Running a simple script with Node*.

Running Node with no arguments plops you in an interactive JavaScript shell:

```
$ node
> console.log('Hello, world!');
Hello, world!
```

Any code you can write in a Node script can be written here. The command interpreter gives a good terminal-orientated user experience and is useful for interactively playing with your code. You do play with your code, don't you? Good!

Running a simple script with Node

Now let's see how to run scripts with Node. It's quite simple. Let's start by referring back to the `help` message:

```
$ node --help
Usage: node [options] script.js [arguments]
```

It's just a script filename and some script arguments, which should be familiar for anyone who has written scripts in other languages.

Creating and editing Node scripts can be done with any text editor that deals with plain text files, such as `vi/Vim`, `Emacs`, `Notepad++`, `jEdit`, `BBEdit`, `TextMate`, or `Komodo`. It's helpful if it's a programmer-oriented editor, if only for the syntax coloring.

For this and other examples in the book, it doesn't really matter where you put the files. However, for neatness' sake, you could start by making a directory named `node-web-dev` in the home directory of your computer, then inside that create one directory per chapter (for example, `chap02` and `chap03`).

First create a text file named `ls.js` with the following content:

```
var fs = require('fs');
var files = fs.readdirSync('.');
for (fn in files) {
  console.log(files[fn]);
}
```

Next, run it by typing the following command:

```
$ node ls.js
app.js
ls.js
```

This is a pale, cheap imitation of the Unix `ls` command (as if you couldn't figure that out from the name). The `readdirSync` function is a close analog to the Unix `readdir` system call (type `man 3 readdir` in a terminal window to learn more) and is used to list the files in a directory.

The script arguments land in a global array named `process.argv` and you can modify `ls.js` as follows to see how this array works:

```
var fs = require('fs');
var dir = '.';
if (process.argv[2]) dir = process.argv[2];
var files = fs.readdirSync(dir);
for (fn in files) {
  console.log(files[fn]);
}
```

And you can run it as follows:

```
$ node ls2.js ~/node/0.10.7/bin
node
node-waf
npm
```

Launching a server with Node

Many scripts that you'll run are server processes. We'll be running lots of these scripts later on. Since we're still in the dual mode of verifying the installation and familiarizing you with using Node, we want to run a simple HTTP server. Let's borrow the simple server script on the Node home page (<http://nodejs.org>).

Create a file named `app.js` containing the following:

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

And run it this way:

```
$ node app.js
Server running at http://127.0.0.1:8124
```

This is the simplest of web servers you can build with Node. If you're interested in how it works, flip forward from *Chapter 4, HTTP Servers and Clients - A Web Application's First Steps* to *Chapter 6, Data Storage and Retrieval*. At the moment just visit `http://127.0.0.1:8124` in your browser to see the **Hello, World!** message.

A question to ponder is why this script did not exit, when `ls.js` exited. In both cases, execution of the script reaches the end of the script; in `app.js`, the Node process does not exit, while in `ls.js` it does. The reason is the presence of active event listeners. Node always starts up an event loop, and in `app.js` the `listen` function creates an event listener which implements the HTTP protocol. This event listener keeps `app.js` running until you do something such as press `Ctrl + C` in the terminal window. In `ls.js` there is nothing that creates a long-running event listener, so when `ls.js` reaches the end of its script, Node will exit.

npm – the Node package manager

Node by itself is a pretty basic system, being a JavaScript interpreter with a few interesting asynchronous I/O libraries. One of the things which makes Node interesting is the rapidly growing ecosystem of third party modules for Node. At the center of that ecosystem is `npm`. The modules can be downloaded at source and assembled manually for use with Node programs. The `npm` command gives us a simpler way; `npm` is the de-facto standard package manager for Node and it greatly simplifies downloading and using these modules. We will talk about `npm` at length in the next chapter.

The sharp-eyed will have noticed that `npm` is already installed via all the installation methods already discussed. In the past, `npm` was installed separately, but today it is bundled with Node.

Now that we have `npm` installed, let's take it for a quick spin. `Hexy` is a utility program for printing hex dumps of files. It serves our purpose right now in giving us something to quickly install and try out.

```
$ npm install -g hexy
npm http GET https://registry.npmjs.org/hexy
npm http 200 https://registry.npmjs.org/hexy
npm http GET https://registry.npmjs.org/hexy/-/hexy-0.2.5.tgz
npm http 200 https://registry.npmjs.org/hexy/-/hexy-0.2.5.tgz
/opt/local/bin/hexy -> /opt/local/lib/node_modules/hexy/bin/hexy_cmd.js
hexy@0.2.5 /opt/local/lib/node_modules/hexy
```

Adding the `-g` flag makes the module available globally, irrespective of which directory it is installed in. It is most useful when the module provides a command-line interface, because `npm` ensures the command is installed correctly for use by all users of the computer.

Depending on how Node is installed for you, that may need to be run with `sudo`.

```
$ sudo npm install -g hexy
```

Once it is installed, you'll be able to run the newly installed program this way:

```
$ hexy --width 12 ls.js
00000000: 7661 7220 6673 203d 2072 6571  var.fs=.req
0000000c: 7569 7265 2827 6673 2729 3b0a  uire('fs');.
00000018: 7661 7220 6669 6c65 7320 3d20  var.files=.
00000024: 6673 2e72 6561 6464 6972 5379  fs.readdirSy
00000030: 6e63 2827 2e27 293b 0a66 6f72  nc('.');.for
0000003c: 2028 666e 2069 6e20 6669 6c65  .(fn.in.file
00000048: 7329 207b 0a20 2063 6f6e 736f  s){...conso
00000054: 6c65 2e6c 6f67 2866 696c 6573  le.log(files
00000060: 5b66 6e5d 293b 0a7d 0a          [fn]);.}
```

Again, we'll be doing a deep dive into `npm` in the next chapter. The `hexy` utility is both a Node library and a script for printing out these old style hex dumps.

Starting Node servers at system startup

Earlier we started a Node server from the command line. While this is useful for testing and development, it's not useful for deploying an application in any normal sense. There are normal practices for starting server processes, which differ for each operating system. Implementing a Node server means starting it similarly to the other background processes (sshd, Apache, MySQL, and so on) using, for example, start/stop scripts.

The Node project does not include start/stop scripts for any operating system. It can be argued that it would be out of place for Node to include such scripts. Instead, Node server applications should include such scripts. The traditional way is that the `init` daemon manages background processes using scripts in the `/etc/init.d` directory. On Fedora and Red Hat, that's still the process, but other operating systems use other daemon managers such as Upstart or `launchd`.

Writing these start/stop scripts is only a part of what's required. Web servers have to be reliable (for example auto-restarting on crashes), manageable (integrate well with system management practices), observable (saving `STDOUT` to logfiles), and so on. Node is more like a construction kit with the pieces and parts for building servers, and is not a complete polished server itself. Implementing a complete web server based on Node means scripting to integrate with the background process management on your OS, implementing the logging features you need, the security practices or defenses against bad behavior, such as denial of service attacks, and much more.

Here are several tools or methods for integrating Node servers with background process management on several operating systems, to implement continuous server presence beginning at system startup. In a moment we'll also do a brief walkthrough of using `forever` on a Debian server. You can run Node as a background daemon on different platforms using the following ways:

- `forever` (<https://github.com/nodejitsu/forever>) is a small command-line Node script which ensures a script will run forever. For a definition of `forever`, Charlie Robbins wrote a blog post (<http://blog.nodejitsu.com/keep-a-nodejs-server-up-with-forever>) about its use.
- `node-init` (<https://github.com/frodwith/node-init>) is a Node script which turns your Node application into an LSB-compliant init script, LSB being a specification of Linux compatibility.

- Ubuntu's Upstart tool (<http://upstart.ubuntu.com/>) can be used alone (http://caolanmcmahon.com/posts/deploying_node_js_with_upstart) or along with monit (<http://howtonode.org/deploying-node-upstart-monit>) to manage a Node server.
- On Mac OS X, one writes a launchd script. Apple has published a guide on implementing launchd scripts at <http://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/Introduction.html>.

To demonstrate a little bit of what's involved, let's use the `forever` tool, along with an LSB-style init script, to implement a little Node server process. The server used to demonstrate this is a Debian-based VPS with Node and `npm` installed in `/usr/local/node/0.10.7`. The following server script is in `/usr/local/app.js` (not the most correct place to install the app, because it's not quite compatible with the LSB-preferred locations for files, but useful for this demo).

```
#!/usr/bin/env node
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World\n');
}).listen(1337);
```



Note the first line of the script carefully. It is a little bit of Unix/POSIX magic, which many readers will recognize, that helps to make the script executable. This executes the script, after first looking up the location of the "node" executable using the `PATH` environment variable. Any script you plan to execute directly should have this first line, plus have the execute permission set.

The `forever` tool is installed as follows:

```
$ sudo npm install -g forever
```

The `forever` script manages background processes. It can restart any processes that crash, send the standard output and error streams to log files, and has several other useful features. It's worth exploring.

The final bit in getting this to work is a shell script, `/etc/init.d/node`, that acts as an init script to manage this server:

```
#!/bin/sh -e
set -e
PATH=/usr/local/node/0.10.7/bin:/bin:/usr/bin:/sbin:/usr/sbin
```

```
DAEMON=/usr/local/app.js
case "$1" in
  start) forever start $DAEMON ;;
  stop)  forever stop  $DAEMON ;;
  force-reload|restart)
    forever restart $DAEMON ;;
  *) echo "Usage: /etc/init.d/node {start|stop|restart|force-reload}"
    exit 1
    ;;
esac
exit 0
```

On Debian you set up an init script with this command:

```
$ sudo /usr/sbin/update-rc.d node defaults
```

This configures your system so that `/etc/init.d/node` is invoked on reboot and shutdown, to start or stop the background process. During boot-up or shutdown. Each init script is executed, and its first argument is either `start` or `stop`. Therefore, when our init script is executed during boot-up or shutdown one of these two lines will be executed:

```
start) forever start $DAEMON ;;
stop)  forever stop  $DAEMON ;;
```

We can run the init script manually:

```
$ sudo /etc/init.d/node start
info: Running action: start
info: Forever processing file: /usr/local/app.js
```

Because our init script uses the `forever` tool, we can ask `forever` the status of all processes it has started:

```
$ sudo forever list
info: Running action: list
info: Forever processes running
  [0] node /usr/local/app.js [16666, 16664]
/home/me/.forever/7rd6.log 0:0:1:24.817
```

With the server process running on your server, you can open it in a browser window to see the **Hello World** message. With it still running and managed by `forever` we have these processes:

```
$ ps ax | grep node
16664 ? Ssl 0:00 node /usr/local/node/0.8.16/bin/forever start
/usr/local/app.js
16666 ? S    0:00 node /usr/local/app.js
```

When you're done playing with this you can shut it down this way:

```
$ sudo /etc/init.d/node stop
info: Running action: stop
info: Forever processing file: /usr/local/app.js
info: Forever stopped process:
    [0] node /usr/local/app.js [5712, 5711] /home/me/.forever/Gtex.log
0:0:0:28.777
$ sudo forever list
info: Running action: list
info: No forever processes running
```

Summary

We learned a lot in this chapter, about installing Node, using its command-line tools, and how to run a Node server. We also breezed past a lot of details that will be covered later in the book, so be patient.

Specifically, we covered:

- Downloading and compiling the Node source code
- Installing Node either for development use in your home directory, or for deployment in system directories
- Installing Node Package Manager (`npm`), the de-facto standard package manager used with Node
- Running Node scripts or Node servers
- What's required to use Node for a reliable background process

Now that we've seen how to set up the basic system, we're ready to start working on implementing applications with Node. First we must learn the basic building block of Node applications, modules, which is the topic of the next chapter.

3

Node Modules

Before writing Node applications we must learn about Node modules and packages. Modules and packages are the building blocks for breaking down your application into smaller pieces.

In this chapter we will:

- Learn what a module is
- Learn about the CommonJS module specification
- Learn how Node finds modules
- Learn about the npm package management system

So, let's get on with it.

Defining a module

Modules are the basic building blocks for constructing Node applications. A Node module encapsulates functions, hiding details inside a well protected container, and exposing an explicitly declared list of functions.

We have already seen modules in action; every JavaScript file we use in Node is itself a module. It's time to see what they are and how they work.

In the `ls.js` example in *Chapter 2, Setting up Node*, we wrote the following code to pull in the `fs` module, giving us access to its functions:

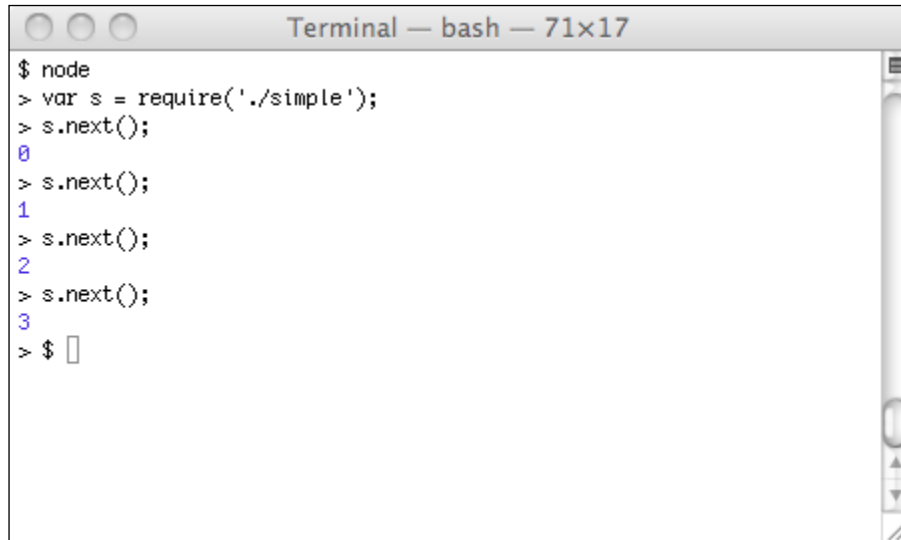
```
var fs = require('fs');
```

The `require` function searches for modules, and loads the module definition into the Node runtime, making its functions available. The `fs` object (in this case) contains the code (and data) exported by the `fs` module.

Let's look at a brief example of this before we start diving into the details. Ponder over this module, `simple.js`:

```
var count = 0;
exports.next = function() { return count++; }
```

This defines an exported function and a local variable. Now let's use it.

A screenshot of a terminal window titled "Terminal — bash — 71x17". The terminal shows the execution of a Node.js script. The prompt is "\$ node". The first command is "> var s = require('./simple');", which is not executed. The subsequent commands are "> s.next();", "> s.next();", "> s.next();", and "> s.next();", which are executed and return the values 0, 1, 2, and 3 respectively. The final prompt is "> \$".

```
$ node
> var s = require('./simple');
> s.next();
0
> s.next();
1
> s.next();
2
> s.next();
3
> $
```

The object `exports` is what is returned from `require('./simple')`. This means each call to `s.next` calls the function `next` in `simple.js`, which returns (and increments) the value of the `count` variable. Hence, each call to `s.next` returns progressively bigger numbers.

The rule is that anything (functions or objects) assigned as a field of `exports` is exported from the module, and objects not assigned to `exports` are not visible to any code outside the module. In this way, Node provides a strong encapsulation of objects inside modules.

This is also how Node avoids the global object problem. The **globals** in a module are actually local to that module, and there are absolutely no truly global objects.

Now that we've got a taste for modules, let's take a deeper look.

Node modules

Node's module implementation is strongly inspired by, but not identical to, the CommonJS module specification (described at the end of this chapter, in the *CommonJS modules* section). The differences between them might only be important if you need to share code between Node and other CommonJS systems. A quick scan of the Modules/1.1.1 spec indicates that the differences are minor, and for our purposes it's enough to just get on with the task of learning to use Node without dwelling too long on the differences.

Node's algorithm for resolving `require(module)`

In Node, modules are stored in files; one module per file. There are several ways to specify module names, and several ways to organize the deployment of modules in the file system. It's quite flexible, especially when used with `npm`, the de-facto standard package manager for Node.

Module identifiers and path names

Generally speaking, the module name is a pathname, but with the file extension removed. That is, when we wrote `require('./simple')` earlier, Node knew to add `.js` to the filename and load in `simple.js`.

Modules whose filenames end in `.js` are of course expected to be written in JavaScript. Node also supports binary code native libraries as Node modules. In this case the file name extension to use is `.node`. It's outside the scope of this book to discuss implementation of a native code Node module, but this gives you enough knowledge to recognize them when you come across them.

Some Node modules are not files in the file system, but are baked into the Node executable. These are the core modules, the ones documented on `nodejs.org`. Their original existence is as the files in the Node source tree, but the build process compiles them into the binary Node executable.

There are three types of module identifiers: relative, absolute, and top-level.

Relative module identifiers begin with `./` or `../` and absolute identifiers begin with `/`. These are identical with POSIX file system semantics with pathnames being relative to the file being executed. That is, a module identifier beginning with `./` is looked for in (or below) the current directory, while one starting with `../` is looked for starting in the parent directory.

Absolute module identifiers, beginning with `/`, obviously are relative to the root of the file system.

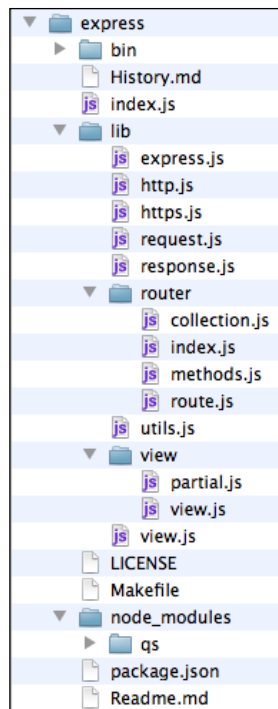
Top-level module identifiers do not begin with `./`, `../`, or `/`, and instead are simply the module name. These modules are stored in one of several directories, such as a `node_modules` directory, or the system-wide directories discussed later.

Local modules within your application

The universe of all possible modules is split neatly into two kinds; those modules that are part of a specific application, and those modules that aren't. Let's begin with the implementation of modules used within your application.

Typically your application will have a directory structure of module files sitting next to each other in the source control system, and then deployed to servers. These modules will know the relative path to their sibling modules within the application, and should use that knowledge to refer to each other, using relative module identifiers.

For example, to help us understand this, let's look at the structure of an existing Node package, the Express web application framework. It includes several modules structured in a hierarchy that the Express developers found to be useful. You can imagine creating a similar hierarchy for applications reaching a certain level of complexity, subdividing the application into chunks larger than a module but smaller than an application. Each subdivided chunk would be implemented as a directory with a few modules in it.



In this example, the most likely relative module reference is to `utils.js`. Depending on the source file which wants to use `utils.js` it would use one of the following `require` statements:

```
var utils = require('./lib/utils');  
var utils = require('./utils');  
var utils = require('../utils');
```

Bundling external dependencies with your application

Modules placed in a `node_modules` directory are accessed using a top-level module identifier such as:

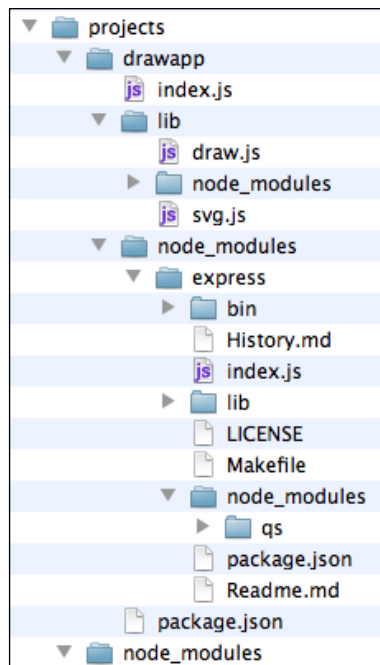
```
var express = require('express');
```

Node searches the `node_modules` directories to find modules. There is not just one `node_modules` directory, but several that are searched by Node. Node starts at the directory of the current module, appends `node_modules`, and searches there for the module being requested. If not found in that `node_modules` directory it moves to the parent directory and tries again, repeatedly moving to parent directories until it reaches the root of the file system.

In the previous example, you'll notice a `node_modules` directory within which there is a directory named `qs`. By being situated in that location, the `qs` module is available to any module inside Express with this code utterance:

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

What if you want to use the Express framework in your application? That's simple; make a `node_modules` directory inside the directory structure of your application, and install the Express framework there:



We can see this in a hypothetical application shown here, `drawapp`. With the `node_modules` directory situated where it is, any module within `drawapp` can access `express` with the code:

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

But those same modules cannot access the `qs` module stashed inside the `node_modules` directory within the `express` module. The search for the `node_modules` directories containing the desired module goes upward in the file system hierarchy, and not into child directories.

Likewise, a module could be installed in `lib/node_modules` and be accessible from `draw.js` or `svg.js` and not accessible from `index.js`. The search for the `node_modules` directories goes upward, and not into child directories.

Node searches upward for the `node_modules` directories, stopping at the first place it finds the module it's searching for. A module reference from `draw.js` or `svg.js` would search this list of directories:

- `/home/david/projects/drawapp/lib/node_modules`
- `/home/david/projects/drawapp/node_modules`
- `/home/david/projects/node_modules`
- `/home/david/node_modules`
- `/home/node_modules`
- `/node_modules`

The `node_modules` directory plays a key role in keeping the Node package management from disappearing into a maze of conflicting package versions. Rather than having one place to put modules and descend into confusion, (as dependencies on conflicting module versions slowly drive you nuts), you can have multiple `node_modules` directories which let you have specific versions of modules in specific places, if needed. Different versions of the same module can live in different `node_modules` directories, and they won't conflict with each other, so long as the `node_modules` directories are situated correctly.

System-wide modules in `NODE_PATH` and elsewhere

The algorithm Node uses to find the `node_modules` directories extends beyond your application source tree. It goes to the root of the file system, and you could have a `/node_modules` directory with a global module repository to satisfy any search for modules.

Node provides an additional mechanism to look in other directories as well. The recommended method is to install modules in the `node_modules` directories, but these other directories can be helpful:

- The first is the `NODE_PATH` environment variable, which is a colon-separated list of directory names, in which to search while looking for modules. On Windows, this list is, of course, semi-colon separated.

- The second is this list of directories:
 - `$HOME/.node_modules`
 - `$HOME/.node_modules`
 - `$PREFIX/lib/node`
- The last directory, `$PREFIX/lib/node`, refers to the location where Node is installed.

Complex modules – modules as directories

A complex module might include several internal modules, data files, template files, documentation, tests, or more. These can be stashed inside a carefully constructed directory structure, which Node will treat as a module satisfying a `require('moduleName')` request. To do so, you place one of the two files in a directory, either a module file named `index.js`, or a file named `package.json`. The `package.json` file will contain data describing the module in a format nearly identical to the `package.json` format defined by the `npm` package manager (described later). The two are compatible with Node, using a very small subset of tags that `npm` recognizes.

Specifically, Node recognizes these fields in `package.json`:

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

The code `require('myAwesomeLibrary')` would find this directory with `package.json`, and load the `/path/to/node_modules/myAwesomeLibrary/lib/awesome.js` file.

If there was no `package.json` file, then Node will instead look for `index.js`, which would load the `/path/to/node_modules/myAwesomeLibrary/index.js` file.

Under either scenario (`index.js` or `package.json`), the complex module with internal modules, and other assets, is easy to implement. Internal modules would be referenced from these modules, making the internal modules an implementation detail, which is not visible outside the package.

Referring back to the Express package structure we looked at earlier, some of the modules will use relative module identifiers to reference other modules inside the package, and you can use a `node_modules` directory to integrate modules developed elsewhere.

Some packages include inner modules that could themselves be useful to other software. For example, the current version of this `openid` module we've been picking on includes a `base64 encode/decode` module that could be useful for other software:

```
var base64 = require('openid/lib/base64').base64;
```

This runs a risk; the `openid` module could change the implementation of its `base64 encode/decode` function, breaking your application. Some packages structured themselves to provide a family of related submodules accessed this way, and provide some guarantee of a stable API for the exposed submodules.

Node package manager

As described in *Chapter 2, Setting up Node*, `npm` is a package management and distribution system for Node. It has become the de-facto standard for distributing modules (packages) for use with Node. Conceptually it's similar to tools like `apt-get` (Debian), `rpm/yum` (Red Hat/Fedora), `MacPorts` (Mac OS X), `CPAN` (Perl), or `PEAR` (PHP). Its purpose is publishing and distributing Node packages over the Internet using a simple command-line interface. With `npm` you can quickly find packages to serve specific purposes, download them, install them, and manage packages you've already installed.

The `npm` package defines a package format for Node largely based on the CommonJS package specification. It uses the same `package.json` file that's supported natively by Node, but with additional fields to build in additional functionality.

The npm package format

An `npm` package is a directory structure with a `package.json` file describing the package. This is exactly what we just referred to as a complex module, except that `npm` recognizes many more `package.json` tags than Node does. The starting point for `npm`'s `package.json` is the CommonJS Packages/1.0 specification. The documentation for `npm`'s `package.json` implementation is accessed using the following command:

```
$ npm help json
```

A basic `package.json` file is as follows:

```
{name: "packageName",  
  version: "1.0",  
  main: "mainModuleName",  
  modules: {  
    "mod1": "lib/mod1",
```

```
    "mod2": "lib/mod2"
  }
}
```

The file is in JSON format, which, as a JavaScript programmer, you should be familiar with.

The most important tags are `name` and `version`. The name will appear in URLs and command names, so choose one that's safe for both. If you desire to publish a package in the public npm repository, it's helpful to check and see if a particular name is already being used at <http://search.npmjs.org> or with the following command:

```
$ npm search packageName
```

The `main` tag is treated the same as we discussed in the previous section on complex modules. It references the module that will be returned when invoking `require('packageName')`. Packages can contain many modules within themselves, and those can be listed in the `modules` list.

Packages can be bundled as tar-gzip (tarballs), especially to send them over the Internet.

A package can declare dependencies on other packages. That way npm can automatically install other modules required by the module being installed. Dependencies are declared as follows:

```
"dependencies":
{
  "foo" : "1.0.0 - 2.x.x",
  "bar" : ">=1.0.2 <2.1.2"
}
```

The `description` and `keyword` fields help people to find the package when searching in an npm repository (<http://search.npmjs.org>). Ownership of a package can be documented in the `homepage`, `author`, or `contributors` fields:

```
"description": "My wonderful packages that walks dogs",
"homepage": "http://npm.dogs.org/dogwalker/",
"author": dogwhisperer@dogs.org
```

Some npm packages provide executable programs meant to be in the user's `PATH`. These are declared using the `bin` tag. It's a map of command names to the script which implements that command. The command scripts are installed into the directory containing the node executable using the name given.

```
bin: {
  'nodeload.js': './nodeload.js',
  'nl.js': './nl.js'
},
```

The `directories` tag describes the package directory structure. The `lib` directory is automatically scanned for modules to load. There are other directory tags for binaries, manuals, and documentation.

```
directories: { lib: './lib', bin: './bin' },
```

The script tags are script commands run at various events in the life cycle of the package. These events include `install`, `activate`, `uninstall`, `update`, and more. For more information about script commands, use the following command:

```
$ npm help scripts
```

This was only a taste of the npm package format; see the documentation (`npm help json`) for more.

Finding npm packages

By default, npm modules are retrieved over the Internet from the public package registry maintained on <http://npmjs.org>. If you know the module name it can be installed simply by typing the following:

```
$ npm install moduleName
```

But what if you don't know the module name? How do you discover the interesting modules?

The website <http://npmjs.org> publishes an index of the modules in that registry, and the <http://search.npmjs.org> site lets you search that index.

The npm package also has a command-line search function to consult the same index:

```
$ npm search mp3
mediatags  Tools extracting for media meta-data tags =coolaj86 util
m4a aac mp3 id3 jpeg exiv xmp
node3p     An Amazon MP3 downloader for NodeJS.      =ncb000gt
```

Of course upon finding a module it's installed as follows:

```
$ npm install mediatags
```

After installing a module you may want to see the documentation, which would be on the module's website. The `homepage` tag in `package.json` lists that URL. The easiest way to look at the `package.json` file is with the `npm view` command, as follows:

```
$ npm view zombie
...
```

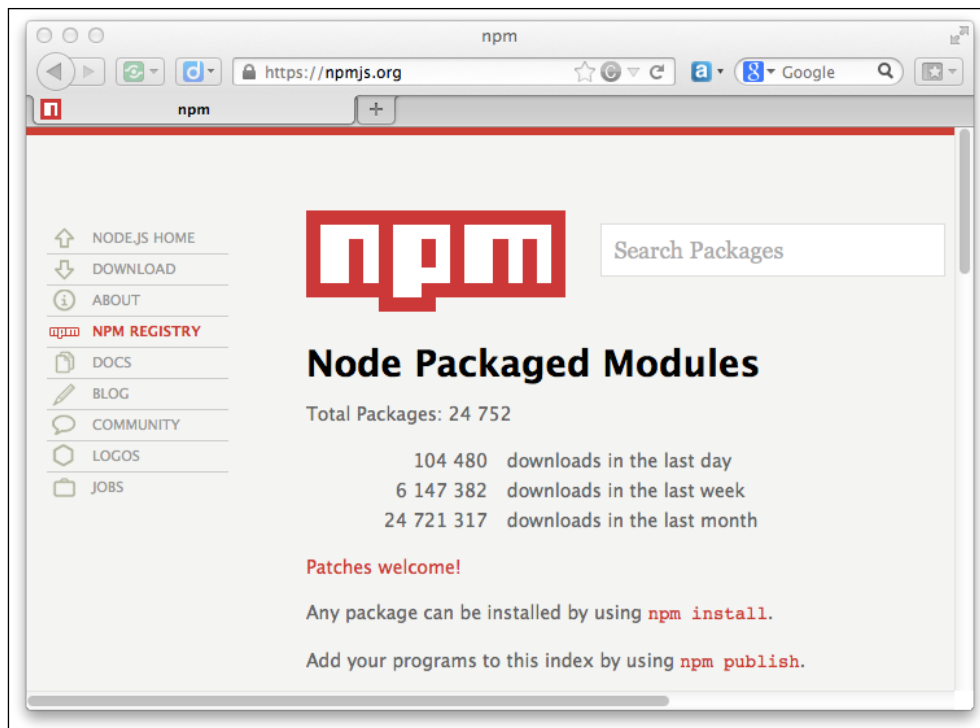
```
{ name: 'zombie',  
  description: 'Insanely fast, full-stack, headless browser testing using  
Node.js',  
  ...  
  version: '1.4.1',  
  homepage: 'http://zombie.labnotes.org/',  
  ...  
}
```

You can use `npm view` to extract any tag from `package.json`, like the following which lets you view just the homepage tag:

```
$ npm view zombie homepage  
http://zombie.labnotes.org/
```

Using the npm commands

The main `npm` command has a long list of subcommands for specific package management operations. These cover every aspect of the life cycle of publishing packages (as a package author), and downloading, using, or removing packages (as an `npm` consumer).



Getting help with npm

Perhaps the most important thing is to learn where to turn to get help. The main help is delivered along with the npm command, accessed by typing the following:

```
$ npm help <command>
```

The npm website (<http://npmjs.org/>) has an FAQ page that is also delivered with the npm software. Perhaps the most important question (and answer) is: "Why does npm hate me?" "npm is not capable of hatred. It loves everyone, even you."

Viewing package information

The npm view command treats the package.json file as data, letting you query that data is using a dot notation for JSON tags. This lets you view the package dependencies:

```
$ npm view google-openid dependencies
{ express: '>= 0.0.1',
  openid: '>= 0.1.1 <= 0.1.1' }
```

The package.json file can include the package repository URL. Therefore, this lets you retrieve that URL, which you can use to retrieve the package source:

```
$ npm view openid repository.url
git://github.com/havard/node-openid.git
$ git clone git://github.com/havard/node-openid.git
Cloning into node-openid...
remote: Counting objects: 253, done.
remote: Compressing objects: 100% (253/253), done.
remote: Total 253 (delta 148), reused 0 (delta 0)
Receiving objects: 100% (253/253), 63.29 KiB, done.
Resolving deltas: 100% (148/148), done.
```

This lets you query the engines value, which describes the versions of Node which can be used to run this module.

```
$ npm view openid engines
node >= 0.6.0
```


Installing an npm package

The `npm install` command makes it easy to install packages upon finding the one of your dreams as follows:

```
$ npm install openid
openid@0.4.2 node_modules/openid
$ ls node_modules/
openid
```

Notice that the package is installed in a local `node_modules` directory. Packages can be installed in other locations either by changing the current directory, or by telling `npm` to make a global install. For example, the following will set up a directory, `/var/www`, where `/var/www/node_modules` stores modules to be shared among several websites:

```
$ cd /var/www
$ npm install openid
openid@0.4.2 node_modules/openid
```

The `npm` package makes a distinction between global mode and local mode. Normally it operates in local mode and installs packages into a local `node_modules` directory next to your application code. In global mode, packages are installed globally, meaning that they're installed into the Node installation (directories in `require.paths`) rather than in a local `node_modules` directory.

The first method to install packages in the global mode is to use the `-g` flag as follows:

```
$ npm install -g openid
openid@0.4.2 /opt/local/lib/node_modules/openid
$ which node
/opt/local/bin/node
```

The installation directory in global mode is based on where Node is installed for you.

The second method for global mode installation is to change the `npm` configuration settings. There are many configuration settings, which we'll discuss in some time, and the relevant one for now is as follows:

```
$ npm set global=true
$ npm get global
true
$ npm install openid
openid@0.4.2 /opt/local/lib/node_modules/openid
```

To learn about all the folders npm uses, enter the following command:

```
$ npm help folders
```

The point of installing a package is to enable a Node program to access the module as the following:

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

What npm does is to help you set up conditions for this to work smoothly. It interacts neatly with Node's module discovery and consumption mechanisms we discussed earlier.

Installing native code modules on Windows

Some Node modules are written with C or C++, meaning they must be compiled to native code. This is handled automatically by npm and it will use node-gyp to compile to native code, and to do so node-gyp depends on a compiler toolchain that includes Python 2.7 and a C/C++ compiler.

On Unix/Linux and Mac OS X systems these dependencies are pretty easy to satisfy. We went over this in *Chapter 2, Setting up Node*, while discussing the installation of Node from source. However, the Python and C/C++ tools on Windows are harder to come by.

A Windows user can see the problem if their system doesn't already have the dependencies installed, by typing this command:

```
C:\> npm install levelup leveledown
```

Sit back and watch the error messages roll by.

The node-gyp project page documents how to install the compiler dependencies, which can be seen at <https://github.com/TooTallNate/node-gyp#installation>.

Installing packages local to a module

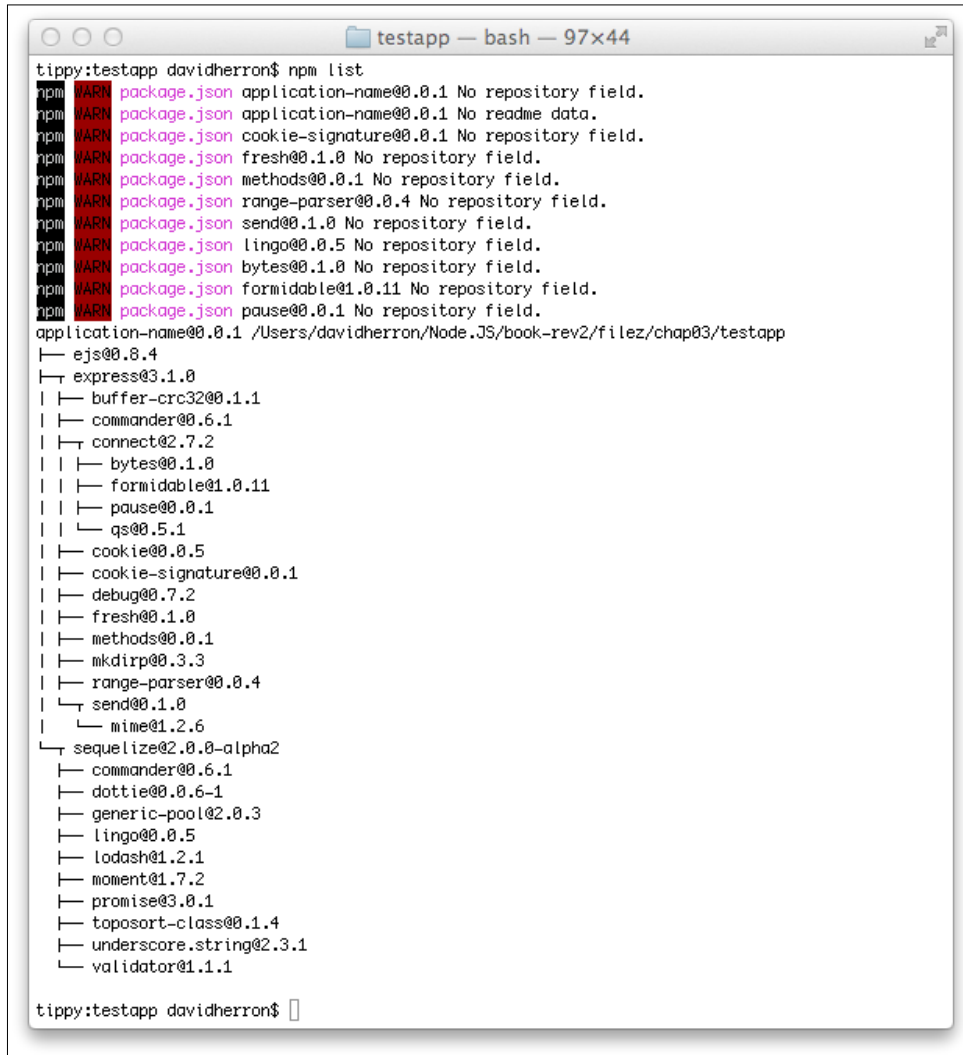
The `node_modules` directories can be anywhere, including inside a module. If you're working on an application, it is helpful to install the module dependencies within the application's `node_modules` directory. Simply do this:

```
$ npm install
```

When used without specifying a module name, it looks for the `package.json` file, and installs any module listed in the `dependencies` tag.

Eliminating duplicate modules installed beneath node_modules

It often comes to pass that the dependent packages themselves have dependencies, and that after installing all dependencies there will be duplicate copies of some modules within the node_modules directory structure. For example:

A terminal window titled 'testapp — bash — 97x44' showing the output of the command 'npm list'. The output lists the installed packages and their dependencies. At the top, there are several 'WARN package.json' messages indicating missing repository and readme fields for various packages. The main list shows a tree structure of dependencies. Notably, 'commander@0.6.1' is listed as a dependency for both 'express@3.1.0' and 'sequelize@2.0.0-alpha2', illustrating duplicate installations.

```
tippy:testapp davidherron$ npm list
npm WARN package.json application-name@0.0.1 No repository field.
npm WARN package.json application-name@0.0.1 No readme data.
npm WARN package.json cookie-signature@0.0.1 No repository field.
npm WARN package.json fresh@0.1.0 No repository field.
npm WARN package.json methods@0.0.1 No repository field.
npm WARN package.json range-parser@0.0.4 No repository field.
npm WARN package.json send@0.1.0 No repository field.
npm WARN package.json lingo@0.0.5 No repository field.
npm WARN package.json bytes@0.1.0 No repository field.
npm WARN package.json formidable@1.0.11 No repository field.
npm WARN package.json pause@0.0.1 No repository field.
application-name@0.0.1 /Users/davidherron/Node.JS/book-rev2/filez/chap03/testapp
├─ ejs@0.8.4
├─ express@3.1.0
│ └─ buffer-crc32@0.1.1
│   └─ commander@0.6.1
│     └─ connect@2.7.2
│       └─ bytes@0.1.0
│         └─ formidable@1.0.11
│           └─ pause@0.0.1
│             └─ qs@0.5.1
│               └─ cookie@0.0.5
│                 └─ cookie-signature@0.0.1
│                   └─ debug@0.7.2
│                     └─ fresh@0.1.0
│                       └─ methods@0.0.1
│                         └─ mkdirp@0.3.3
│                           └─ range-parser@0.0.4
│                             └─ send@0.1.0
│                               └─ mime@1.2.6
│                                 └─ sequelize@2.0.0-alpha2
│                                   └─ commander@0.6.1
│                                     └─ dottie@0.0.6-1
│                                       └─ generic-pool@2.0.3
│                                         └─ lingo@0.0.5
│                                           └─ lodash@1.2.1
│                                             └─ moment@1.7.2
│                                               └─ promise@3.0.1
│                                                 └─ toposort-class@0.1.4
│                                                   └─ underscore.string@2.3.1
│                                                     └─ validator@1.1.1
└─ tippy:testapp davidherron$
```

What's happened is both Express and Sequelize are using the same version of the commander package.

To eliminate the duplicate modules, type the following:

```
$ npm dedupe
```

```

testapp — bash — 97x43
tippy:testapp davidherron$ npm list
npm WARN package.json application-name@0.0.1 No repository field.
npm WARN package.json application-name@0.0.1 No readme data.
npm WARN package.json cookie-signature@0.0.1 No repository field.
npm WARN package.json fresh@0.1.0 No repository field.
npm WARN package.json methods@0.0.1 No repository field.
npm WARN package.json range-parser@0.0.4 No repository field.
npm WARN package.json send@0.1.0 No repository field.
npm WARN package.json lingo@0.0.5 No repository field.
npm WARN package.json bytes@0.1.0 No repository field.
npm WARN package.json formidable@1.0.11 No repository field.
npm WARN package.json pause@0.0.1 No repository field.
application-name@0.0.1 /Users/davidherron/Node.JS/book-rev2/filez/chap03/testapp
├── commander@0.6.1
├── ejs@0.0.4
├── express@3.1.0
│   ├── buffer-crc32@0.1.1
│   ├── connect@2.7.2
│   │   ├── bytes@0.1.0
│   │   ├── formidable@1.0.11
│   │   ├── pause@0.0.1
│   │   ├── qs@0.5.1
│   │   ├── cookie@0.0.5
│   │   ├── cookie-signature@0.0.1
│   │   ├── debug@0.7.2
│   │   ├── fresh@0.1.0
│   │   ├── methods@0.0.1
│   │   ├── mkdirp@0.3.3
│   │   ├── range-parser@0.0.4
│   │   ├── send@0.1.0
│   │   └── mime@1.2.6
│   └── sequelize@2.0.0-alpha2
│       ├── dottie@0.0.6-1
│       ├── generic-pool@2.0.3
│       ├── lingo@0.0.5
│       ├── lodash@1.2.1
│       ├── moment@1.7.2
│       ├── promise@3.0.1
│       ├── topological-sort@0.1.4
│       ├── underscore.string@2.3.1
│       └── validator@1.1.1
└── tippy:testapp davidherron$

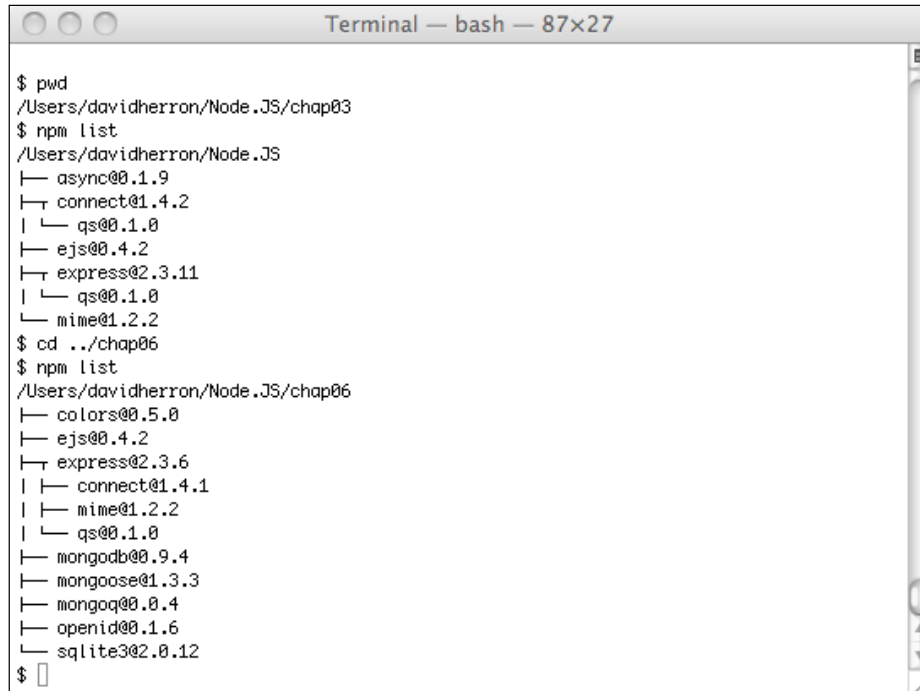
```

The dedupe command moves common modules further up the node_modules directory tree, so that one instance of a given module will serve every module in the tree. In this example, look carefully at what happens to the commander module.

Listing the currently installed packages

The npm list command lists the installed packages, as we just saw, based on a search from the current directory. Remember that Node searches for modules starting at the current directory of the code being executed. Therefore, the installed packages are relatively based on your current directory, depending on the content of the node_modules directories above the current directory.

For example, notice how the listed modules change based on which directory you are in:

A terminal window titled "Terminal — bash — 87x27" showing the output of the 'npm list' command in two different directories. The first directory is /Users/davidherron/Node.JS/chap03, and the second is /Users/davidherron/Node.JS/chap06. The output is shown in a tree structure with indentation and arrows indicating dependencies.

```
$ pwd
/Users/davidherron/Node.JS/chap03
$ npm list
/Users/davidherron/Node.JS
├─ async@0.1.9
├─ connect@1.4.2
│ └─ qs@0.1.0
├─ ejs@0.4.2
├─ express@2.3.11
│ └─ qs@0.1.0
└─ mime@1.2.2
$ cd ../chap06
$ npm list
/Users/davidherron/Node.JS/chap06
├─ colors@0.5.0
├─ ejs@0.4.2
├─ express@2.3.6
│ └─ connect@1.4.1
│   └─ mime@1.2.2
│     └─ qs@0.1.0
├─ mongodb@0.9.4
├─ mongoose@1.3.3
├─ mongoq@0.0.4
├─ openid@0.1.6
└─ sqlite3@2.0.12
$
```

By default, the list is shown in a tree structure, which isn't terribly useful (as data) to other commands, as shown in the previous screenshot. The parseable configuration setting can make the output usable as data as follows:

```
$ npm set parseable=true
$ npm list
/home/david/Node/chap06
/home/david/Node/chap06/node_modules/ejs
/home/david/Node/chap06/node_modules/express
/home/david/Node/chap06/node_modules/express/node_modules/connect
/home/david/Node/chap06/node_modules/express/node_modules/mime
/home/david/Node/chap06/node_modules/express/node_modules/qs
/home/david/Node/chap06/node_modules/mongodb
/home/david/Node/chap06/node_modules/mongoose
/home/david/Node/chap06/node_modules/sqlite3
```

Package scripts

The npm package allows package scripts to automatically run at various times in the package life cycle. Currently there are four lifecycle events: `test`, `start`, `stop`, and `restart`. There is further documentation available with this npm command:

```
$ npm help run-scripts
```

An npm package can include tests which are run as follows:

```
$ npm test <packageName>
```

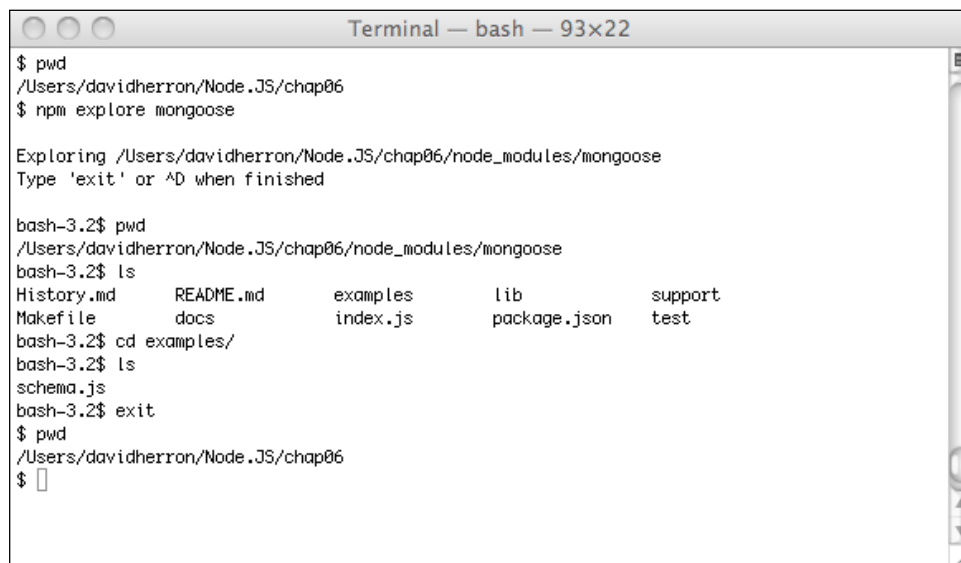
The `start`, `stop`, and `restart` life cycle events don't have a defined meaning. An obvious use is starting or stopping daemon processes associated with the package.

The default for the `start` script is to do the equivalent of:

```
$ node server.js
```

Editing and exploring installed package content

The npm package includes a pair of commands to let you look at, or change, package content. For example, you could use this during development to read the package source (say, to understand what it's doing), look in the package examples directory, ensure Node is finding the expected/correct version of the module, or make modifications to test patches. For example:

A screenshot of a macOS Terminal window titled "Terminal — bash — 93x22". The terminal shows the following commands and output:

```
$ pwd
/Users/davidherron/Node.JS/chap06
$ npm explore mongoose

Exploring /Users/davidherron/Node.JS/chap06/node_modules/mongoose
Type 'exit' or ^D when finished

bash-3.2$ pwd
/Users/davidherron/Node.JS/chap06/node_modules/mongoose
bash-3.2$ ls
History.md      README.md      examples       lib            support
Makefile       docs          index.js      package.json  test
bash-3.2$ cd examples/
bash-3.2$ ls
schema.js
bash-3.2$ exit
$ pwd
/Users/davidherron/Node.JS/chap06
$
```

The sequence starts in the user's regular shell, in the directory containing the application source. As the command output implies, the `explore` command spawns a subshell whose current directory is the location where the module is installed. Upon typing the `explore` command, the user's shell is in a new directory. What actually happened is that another child shell was spawned in that new directory. Typing `exit` or pressing `Ctrl + D` ends the subshell, returning you to your login shell.

You can do anything you like in the subshell and it won't modify the environment of the parent shell process. Upon exiting the subshell, the user is returned to their login shell.

You can edit files while browsing the package if you like. If you do, the package may need to be rebuilt as follows:

```
$ npm rebuild mongoose
mongoose@3.5.4 /home/david/Node/chap06/node_modules/mongoose
```

Updating outdated packages you've installed

The coder codes, updating their package, leaving you in their dust unless you keep up.

To find out if your installed packages are out of date, use the following command:

```
$ npm outdated
less@1.3.3 node_modules/less current=1.3.1
gdata-js@2.0.1 node_modules/gdata-js current=0.0.4
consolidate@0.7.0 node_modules/consolidate current=0.5.0
watchr@2.3.3 node_modules/watchr current=2.3.0
rsyncwrapper@0.0.10 node_modules/rsyncwrapper current=0.0.8
```

This shows the current installed version, as well as the current version in the `npm` repository. Updating the outdated packages is very simple:

```
$ npm update express
connect@1.4.1 ./node_modules/express/node_modules/connect
mime@1.2.2 ./node_modules/express/node_modules/mime
qs@0.1.0 ./node_modules/express/node_modules/qs
express@2.3.6 ./node_modules/express
```

Uninstalling an installed npm package

It may come to pass that the package of your dreams turns into a nightmare, and even if it does not, there are plenty of reasons to remove installed packages. For example, you might want to clean up the installed files to start from a clean slate, or a package may no longer be a dependency in a project. This can be done as follows:

```
$ npm list
/home/david/Node
└─ openid@0.4.2
$ npm uninstall openid
$ npm list
/home/david/Node
(empty)
```

Developing and publishing npm packages

Now that we have a good idea of how to use npm let's get to the other end of the process and look at how to develop npm packages. Some of the npm commands serve the development process.

The first step is creating the `package.json` file, and the `npm init` command helps you create the initial version. It interrogates you with the questions required to fill in the `package.json` file shown as follows, and quickly helps you create something like the following:

```
{
  "author": "I.M. Awesome <awesome@example.com>",
  "name": "tmod",
  "description": "Test Module",
  "version": "0.0.1",
  "repository": {
    "url": ""
  },
  "engines": {
    "node": ">0.4.1"
  },
  "dependencies": {},
  "devDependencies": {}
}
```

The next step is obviously to create the package source. The npm package doesn't have any way to help you with this. You are the coder so you do the coding. Just make sure to keep updating the `package.json` file as you add things such as command line tools, or dependencies, to the package.

The npm package does have a couple of commands you'll be using while developing the package. One of these commands is `npm link`; a lighter-weight method of installing packages. The difference between this and `npm install` is that `npm link` simply sets up a symbolic link to your source directory, and you can freely edit package files without having to repack and update the package on every change. You can iteratively work on the package, and test it, without having to continually rebuild.

Using `npm link` is a two-step process, where first you link your project into the Node installation as follows:

```
$ cd tmod
$ npm link
/opt/local/lib/node_modules/tmod -> /home/david/Node/chap03/tmod
```

In the second step, you link that package into your application:

```
$ npm link tmod
../node_modules/tmod -> /opt/local/lib/node_modules/tmod ->
/home/david/Node/chap03/tmod
```

The arrows (`->`) show you the symbolic link chain that's set up by these commands.

The `npm install` command has a couple of modes that are useful during development. The first is that, if executed in the root of a package directory, it installs the current directory and dependencies into the local `node_modules` directory.

The second is to install tarballs, either from a local file or over the network from a URL. Most source code control systems support a URL providing a tarball (compressed tar file) of the source tree. For example, the downloads page on GitHub projects gives a URL like this one:

```
$ npm install https://github.com/havard/node-openid/tarball/v0.1.6
openid@0.1.6 ../node_modules/openid
```

When you're satisfied that your package works, you might want to publish it in the public npm repository so others can use it.

The first step is to register an account with the npm repository. It's done by running the `npm adduser` command, which asks you a series of questions to establish a username, password, and an e-mail address:

```
$ npm adduser
Username: my-user-name
Password:
Email: me@example.com
```

After this step, run the `npm publish` command in the root directory of your package:

```
$ npm publish
```

If all has gone well, after running `npm publish`, you can go to <http://search.npmjs.org> and search for the package. It should show up pretty quickly.

The `npm unpublish` command, as the name implies, removes the package from the npm repository.

npm configuration settings

We've already touched on npm configuration earlier, with global mode versus local mode. There are a number of other settings to fine-tune npm behavior. Let's first look at the ways to make configuration settings.

First is the `npm set` and `npm get` commands, or:

```
npm config set <key> <value> [--global]
npm config get <key>
npm config delete <key>
npm config list
npm config edit
npm get <key>
npm set <key> <value> [--global]
```

For example:

```
$ npm set color true
$ npm set global false
$ npm config get color
true
$ npm config get global
false
```

Environment variables can be used to set configuration values. Any variables which start with `NPM_CONFIG_` are interpreted as configuration values. For example, the variable `NPM_CONFIG_GLOBAL` will set the `global` configuration value.

Configuration values can be put into configuration files:

- `$HOME/.npmrc`
- `<Node Install Directory>/etc/npmrc`

The configuration file contains the `name=value` pairs as follows. This file is updated by the `npm config set` command that we just discussed:

```
$ cat ~/.npmrc
global = false
color = true
```

Package version strings and ranges

Node doesn't know anything about version numbers. It knows about modules, and can treat a directory structure as if it were a module. It has a fairly rich system of looking for modules, but it doesn't know version numbers. The npm package however knows about version numbers. It uses the Semantic Versioning model (see further), and as we've seen, you can install modules over the Internet, query for out-of-date modules, and update them with npm. All of this is version controlled, so let's take a closer look at the things npm can do with version numbers and version tags.

Earlier we used `npm list` to list installed packages, and the listing includes version numbers of installed packages. If instead you wish to see the version number of a specific module, type the following command:

```
$ npm view express version
3.0.6
```

Whenever npm commands take a package name, you can append a version number or version tag to the package name. This way you can deal with specific package versions if needed; for example, if you've tested and qualified your application against a specific version in a staging environment, you can ensure that the same version is used in the deployment environment:

```
$ npm install express@2.3.1
mime@1.2.2 ./node_modules/express/node_modules/mime
connect@1.5.1 ./node_modules/express/node_modules/connect
qs@0.2.0 ./node_modules/express/node_modules/qs
express@2.3.1 ./node_modules/express
```

The npm package has a **tag** concept that might be used to let package users track the **stable** or **test** version of a package. It is used, as shown, to install the latest stable release of the `sax` package:

```
$ npm install sax@stable
```

Tag names are arbitrary and are not required to be used by package authors. The package author designates the meaning of the tag names they use.

Packages list dependencies to other packages, in their `package.json` file, which you can view in this way:

```
$ npm view mongoose dependencies
{ hooks: '0.2.1',
  mongodb: '1.2.8',
  ms: '0.1.0',
  sliced: '0.0.3',
  muri: '0.1.0' }
```

```
$ npm view express dependencies
{ connect: '2.7.2',
  commander: '0.6.1',
  'range-parser': '0.0.4',
  mkdirp: '0.3.3',
  cookie: '0.0.5',
  'buffer-crc32': '0.1.1',
  fresh: '0.1.0',
  methods: '0.0.1',
  send: '0.1.0',
  'cookie-signature': '0.0.1',
  debug: '*' }
```

The package dependencies is the way npm knows which additional modules to install. When installing a module, it looks at the dependencies and downloads any which are currently not installed.

While this will be straightforward and unsurprising to anybody who has dealt with software at any time, there is a sound model behind the scenes. The npm author used the Semantic Versioning specifications at <http://semver.org> to guide the npm version numbering system. It is as follows:

- Version strings are normally integers arranged as X.Y.Z; X is the major version, Y is the minor version, and Z is the patch (for example, 1.2.3).
- The version string can have arbitrary text appended immediately after the patch number for what are called **special versions** (for example, 1.2.3beta1).

- Comparing version strings is not a string comparison, but a numerical comparison of the X, Y, and Z values. For example, $1.9.0 < 1.10.0 < 1.11.3$. Furthermore, $1.0.0\text{beta1} < 1.0.0\text{beta2} < 1.0.0$.
- Compatibility is documented through a version numbering convention:
 - Packages with the major version 0 ($X = 0$) are completely unstable and can change any API at any time.
 - The patch number (Z) like version 1.2.3 to version 1.2.4, must be incremented when the only change is backwards-compatible bug fixes.
 - The minor number (Y) must be incremented when backwards-compatible functionality is introduced (for example, a new function, and all other functions remain compatible).
 - The major number (X) must be incremented when incompatible changes are made.

CommonJS modules

Node's module system is based on the the CommonJS module system (<http://www.commonjs.org/>). While JavaScript is a powerful language with several advanced modern features (such as objects and closures), it lacks a standard object library to facilitate building applications. CommonJS aims to fill that gap with both a convention for implementing modules in JavaScript, and a set of standard modules.

The `require` function takes a module identifier and returns the API exported by the module. If a module requires other modules, those modules are loaded as well. Modules are contained in one JavaScript file, and CommonJS doesn't specify how the module identifier is mapped into a filename.

Modules provide a simple mechanism for encapsulation to hide implementation details while exposing an API. Module content is the JavaScript code, which is treated as if it were written as follows:

```
(function() { ... contents of module file ... })();
```

This encapsulates (hides) every top-level object in the module within a private namespace that other code cannot access. This is how the Global Object problem is resolved (more on that shortly).

The exported module API is the object returned by the `require` function. Inside the module it's implemented with a top-level object named `exports` whose fields contain the exported API. To expose a function or object from the module, simply assign it into the `exports` object.

Demonstrating module encapsulation

That was a lot of words, so let's do a quick example. Create a file named `module1.js` containing the following:

```
var A = "value A";
var B = "value B";
exports.values = function() {
  return { A: A, B: B };
}
```

Then create a file named `module2.js` containing the following:

```
var util = require('util');
var A = "a different value A";
var B = "a different value B";
var m1 = require('./module1');
util.log('A='+A+' B='+B+' values='+util.inspect(m1.values()));
```

Then run it as follows (you must have Node already installed):

```
$ node module2.js
19 May 21:36:30 - A=a different value A B=a different value B
values={ A: 'value A', B: 'value B' }
```

This artificial example demonstrates the encapsulation of the values in `module1.js`, from those in `module2.js`. The `A` and `B` values in `module1.js` don't overwrite `A` and `B` in `module2.js`, because they're encapsulated within `module1.js`. Values encapsulated within a module can be exported, such as the `.values` function in `module1.js`.

The Global Object problem mentioned earlier has to do with those variables that are outside the functions, putting them in the global context. In web browsers, there is a single global context, and it causes a lot of problems if one JavaScript script steps on the global variables used in another script. Each Node module has its own private global context, making it safe to share variables between functions within a module, without danger of interfering with global variables in other modules.

Summary

We learned a lot in this chapter about modules and packages for Node. Specifically, we covered:

- Implementing modules and packages for Node
- Managing installed modules and packages
- How Node locates modules

Now that we've learned about modules and packages, we're ready to use them to build applications, which is the topic of the next chapter.

4

HTTP Servers and Clients – A Web Application's First Steps

Now that we've learned about Node modules, it's time to put this knowledge to work in building a simple Node web application. In this chapter, we'll keep the application simple, enabling us to explore three different application frameworks for Node. In later chapters, we'll do some more complex applications, but before we can walk we must learn to crawl.

Sending and receiving events with EventEmitter

EventEmitters are one of the core idioms of Node, because many of the core modules are `EventEmitters`, and also because `EventEmitters` make an excellent skeleton to implement asynchronous programming on. `EventEmitters` have nothing to do with web application development, but they are so much a part of the woodwork that you may skip over their existence. It is with the HTTP classes that we will get our first taste of `EventEmitters`.

In this chapter we'll be working with the `HTTPServer` and `HTTPClient` objects. Both of them are subclasses of `EventEmitter` and rely on it to send events for each step of the HTTP protocol. Understanding `EventEmitter` will help you understand not only these objects but many other objects in Node.

The `EventEmitter` object is defined in the `events` module of Node. Directly using the `EventEmitter` class means performing `require('events')`, but this is not required, except for cases where the program needs to be a subclass of `EventEmitter`.

Create a file named `pulser.js` that will contain the following code. It shows both the sending as well as the receiving events while directly using the `EventEmitter` class:

```
var events = require('events');
var util   = require('util');
function Pulser() {
  events.EventEmitter.call(this);
}
util.inherits(Pulser, events.EventEmitter);
Pulser.prototype.start = function() {
  var self = this;
  this.id = setInterval(function() {
    util.log('>>>> pulse');
    self.emit('pulse');
    util.log('<<<< pulse');
  }, 1000);
}
```

This defines a `Pulser` class, which inherits from `EventEmitter` (using `util.inherits`). Its purpose is to send timed events, once a second, to any listeners. The `start` method uses `setInterval` to kick off repeated callback execution, scheduled for every second, and call `emit` to send the pulse events to any listeners.

This could be a standalone module for any application that needs timer events at regularly scheduled intervals.

Now let's see how to use the `Pulser` object by adding the following code to `pulser.js`:

```
var pulser = new Pulser();
pulser.on('pulse', function() {
  util.log('pulse received');
});
pulser.start();
```

Here we create a `Pulser` object, and consume its pulse events. Calling `pulser.on('pulse')` sets up connections for the pulse events to invoke the callback function. It then calls the `start` method to get the process going.

Enter this into a file and name the file `pulser.js`, run it, and you should see the following output:

```
$ node pulser.js
23 May 20:30:20 - >>>> pulse
23 May 20:30:20 - pulse received
23 May 20:30:20 - <<<< pulse
```

```
23 May 20:30:21 - >>>> pulse
23 May 20:30:21 - pulse received
23 May 20:30:21 - <<<< pulse
...
```

EventEmitter theory

The EventEmitter events have names, and can be anything which makes sense to you. You can define as many event names as you like. Event names are defined simply by calling `.emit` with the event name. There's nothing formal to do, no registry of event names. Simply making a call to `.emit` is enough to define an event name.

By convention, the event name `error` indicates errors.

An object sends events using the `.emit` function. Events are sent to any listeners that have registered to receive events from the object. The program registers to receive an event by calling that object's `.on` method, giving the event name and an event handler function.

Often it is required to send data along with an event. To do so, simply add the data as arguments to the `.emit` call, as follows:

```
self.emit('eventName', data1, data2, ..);
```

When the program receives that event, the data appears as arguments to the callback function. Your program would listen to such an event as follows:

```
emitter.on('eventName', function(data1, data2, ..) {
  // act on event
});
```

HTTP server applications

The HTTP server object is the foundation of all Node web applications. The object itself is very close to the HTTP protocol, and its use requires knowledge of that protocol. In most cases, you'll be able to use an application framework like Express that hides the HTTP protocol details, allowing the programmer to focus on business logic.

We already saw a simple HTTP server application in *Chapter 2, Setting up Node*, as follows:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
```

```
res.end('Hello, World!\n');
}).listen(8124, '127.0.0.1');
console.log('Server running at http://127.0.0.1:8124');
```

The `http.createServer` function creates an `http.Server` object. Because it is an `EventEmitter`, this could be written another way to make it a little more explicit:

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

The request event takes a function, which receives request and response objects. The request object has data from the web browser, while the response object is used to gather the data to be sent in the response. The `listen` function causes the server to start listening and arranging to dispatch an event for every request arriving from a web browser.

This application isn't very useful, as it returns `Hello World` no matter what was requested from the browser. Real web applications of course give varying responses for different URLs and HTTP verbs.

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

```
var http = require('http');
var util = require('util');
var url = require('url');
var os = require('os');
var server = http.createServer();
server.on('request', function(req, res) {
  var requrl = url.parse(req.url, true);
  if (requrl.pathname === '/') {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end(
      ["<html><head><title>Hello, world!</title></head>",
       "<body><h1>Hello, world!</h1>",
       "<p><a href='/osinfo'>OS Info</a></p>",
       "</body></html>"]
      .join('\n')
    );
  } else if (requrl.pathname === "/osinfo") {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end(
```

```

["<html><head>",
 "<title>Operating System Info</title></head>",
 "<body><h1>Operating System Info</h1>",
 "<table>",
 "<tr><th>TMP Dir</th><td>{tmpdir}</td></tr>",
 "<tr><th>Host Name</th><td>{hostname}</td></tr>",
 "<tr><th>OS Type</th>",
 "<td>{type} {osplat} {osarch} {osrelease}</td></tr>",
 "<tr><th>Uptime</th>",
 "    <td>{uptime} {loadavg}</td></tr>",
 "<tr><th>Memory</th>",
 "    <td>total: {totalmem} free:{freemem}</td></tr>",
 "<tr><th>CPU's</th>",
 "    <td><pre>{cpudata}</pre></td></tr>",
 "<tr><th>Network</th>",
 "    <td><pre>{netdata}</pre></td></tr>",
 "</table>",
 "</body></html>"]
.join('\n')
.replace("{tmpdir}", os.tmpDir())
.replace("{hostname}", os.hostname())
.replace("{type}", os.type())
.replace("{osplat}", os.platform())
.replace("{osarch}", os.arch())
.replace("{osrelease}", os.release())
.replace("{uptime}", os.uptime())
.replace("{loadavg}", util.inspect(os.loadavg()))
.replace("{totalmem}", os.totalmem())
.replace("{freemem}", os.freemem())
.replace("{cpudata}", util.inspect(os.cpus()))
.replace("{netdata}",
        util.inspect(os.networkInterfaces()))
);
} else {
    res.writeHead(404, {'Content-Type': 'text/plain'});
    res.end("bad URL " + req.url);
}
});
server.listen(8124);
console.log('listening to http://localhost:8124');

```

To run it type the following command:

```
$ node server.js
```

```
listening to http://localhost:8124
```

This application is meant to be similar to PHP's `sysinfo` function. Node's `os` module is consulted to give information about the server. This example could easily be extended to gather other pieces of data about the server.

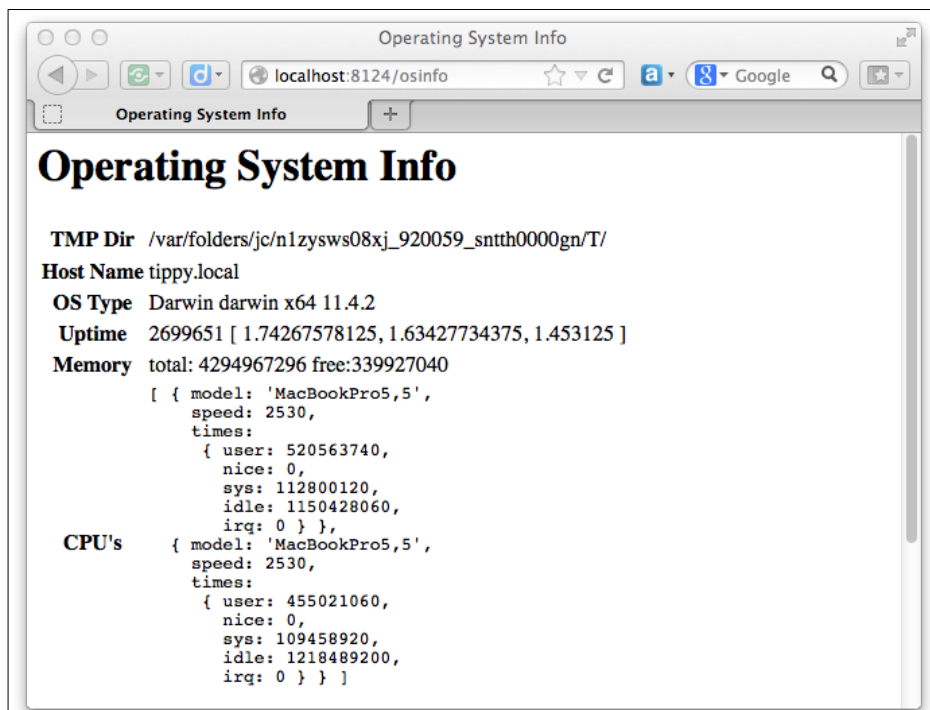
A central part of any web application is the method of routing requests to request handler functions. The `request` object has several pieces of data attached to it, two of which are useful for routing requests — the `request.url` and `request.method` fields.

In `server.js` we consult the `request.url` data to determine which page to show, after parsing (using `url.parse`) to ease the digestion process.

Some web applications care about the HTTP verb (GET, DELETE, POST, and so on) used and must consult the `request.method` field of the request object. For example, POST is frequently used for HTML form processing.

The pathname portion of the request URL is used to dispatch the handler code. While this routing method, based on simple string comparison, will work for a small application, it'll quickly become unwieldy. Larger applications will use pattern matching to use part of the request URL to select the request handler function, and other parts to extract request data out of the URL. We'll see this in action while looking at Express later, in the *Getting started with Express* section.

If the request URL is not recognized, the server sends back an error page using a 404 result code. The result code informs the browser of the status of the request, wherein a 200 code says everything is fine, and a 404 code means the requested page doesn't exist. There are, of course, many other HTTP response codes, each with their own meaning.



HTTP Sniffer – listening to the HTTP conversation

The events emitted by the HTTP Server object can be used for additional purposes beyond the immediate task of delivering a web application. The following code demonstrates a useful module which listens to all the HTTP Server events. It could be a useful debugging tool, which also demonstrates how HTTP Server objects operate.

Node's HTTP Server object is an EventEmitter and the HTTP Sniffer simply listens to every server event, printing out information pertinent to each event.

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

```
var util = require('util');
var url = require('url');
exports.sniffOn = function(server) {
  server.on('request', function(req, res) {
    util.log('e_request');
    util.log(reqToString(req));
  });
  server.on('close', function(errno) {
    util.log('e_close errno='+ errno);
  });
  server.on('checkContinue', function(req, res) {
    util.log('e_checkContinue');
    util.log(reqToString(req));
    res.writeContinue();
  });
  server.on('upgrade', function(req, socket, head) {
    util.log('e_upgrade');
    util.log(reqToString(req));
  });
  server.on('clientError', function() {
    util.log('e_clientError');
  });
}
var reqToString = exports.reqToString = function(req) {
  var ret = 'request ' + req.method + ' ' +
    req.httpVersion + ' ' + req.url + '\n';
  ret += JSON.stringify(url.parse(req.url, true)) + '\n';
  var keys = Object.keys(req.headers);
  for (var i = 0, l = keys.length; i < l; i++) {
    var key = keys[i];
```

```
    ret += i + ' ' + key + ': ' + req.headers[key] + '\n';
  }
  if (req.trailers)
    ret += req.trailers + '\n';
  return ret;
}
```

That was a lot of code! But the key to it is the `sniffOn` function. When given an HTTP Server function, it uses the `.on` function to attach listener functions that print data about each event emitted by the HTTP Server object. It gives a fairly detailed trace of HTTP traffic on an application.

In order to use it, simply insert this code just before the `listen` function:

```
require('./httpsniffer').sniffOn(server);
server.listen(8124);
console.log('listening to http://localhost:8124');
```

With this in place, run the following on the server:

```
$ node server.js
```

You can visit `http://localhost:8124/` in your browser and see the following console output. Notice that two requests are made, one for `/` and one for `/favicon.ico`. The **favicon** is that little image some browsers show to help you brand your website. The server we're using at this moment doesn't support this file, but we'll see later how to implement it.

```
$ node server.js
6 Apr 21:14:38 - e_request
6 Apr 21:14:38 - request GET 1.1 /
{"search":"","query":{"},"pathname":"/","href":"/"}
0 host: localhost:8124
1 user-agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_7; en-us)
AppleWebKit/533.20.25 (KHTML, like Gecko) Version/5.0.4 Safari/533.20.27
2 accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,image/png,*/*;q=0.5
3 cache-control: max-age=0
4 accept-language: en-us
5 accept-encoding: gzip, deflate
6 connection: keep-alive
6 Apr 21:14:39 - e_request
6 Apr 21:14:39 - request GET 1.1 /favicon.ico
```

```
{"search":"","query":{},"pathname":"/favicon.ico","href":"/favicon.ico"}
0 host: localhost:8124
1 user-agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_7; en-us)
AppleWebKit/533.20.25 (KHTML, like Gecko) Version/5.0.4 Safari/533.20.27
2 referer: http://localhost:8124/
3 cache-control: max-age=0
4 accept: */*
5 accept-language: en-us
6 accept-encoding: gzip, deflate
7 connection: keep-alive
```

You now have a tool for snooping on HTTP Server events. This simple technique prints a detailed log of event data and the pattern can be used for any EventEmitter object. You can use this technique as a way to inspect the actual behavior of EventEmitter objects in your program.

Web application frameworks

The HTTP Server object is very close to the HTTP protocol. While this is powerful in the same way that driving a stick shift car gives you better control, the typical web application programming is better done at a higher level. It's better to abstract away the HTTP details and concentrate on your application.

The Node developer community has developed quite a few modules to help with different aspects of abstracting away HTTP protocol details. A good list of these are on the Node modules wiki at <https://github.com/joyent/node/wiki/modules#wiki-web-frameworks>.

Routers assist with routing incoming HTTP requests to handler functions. Some of these behave similarly to other application frameworks like Rails or Sinatra.

Static file servers perform the normal function of serving up a directory hierarchy of files.

The **Frameworks** provide support for developing applications. Some are inspired by other frameworks like Django, Rails, WebMachine, or CakePHP. Most provide some kind of **Model View Controller (MVC)** functionality. Express appears to be the most popular framework for Node.

Middleware operates in the middle space between the reception of an HTTP request and its dispatch to a request handler function.

One reason to use a web framework is that they often provide all the usual best practices that have come from web application development over 20 years. They have been listed as follows:

- Providing a page for bad URLs (the 404 page)
- Screening URLs and forms for any injected scripting attacks
- Supporting the use of cookies to maintain sessions
- Logging requests for both usage tracking and debugging
- Authentication
- Handling static files, such as images, CSS, JavaScript, or HTML
- Providing cache control headers for caching proxies
- Limiting things such as page size, or execution time

Web frameworks help you invest your time in the task without getting lost in the details of implementing HTTP protocol. Abstracting away details is a time honored way for programmers to be more efficient. This is especially true when using a library or framework, providing pre-packaged functions that take care of the details.

Getting started with Express

Express is a web application framework built upon **Connect** (a middleware framework). This means that the focus of Express is on constructing an application that includes providing a template system, whereas the focus of Connect is on web server infrastructural support.

The home page for Connect is <http://senchalabs.github.com/connect/>.

The home page for Express is <http://expressjs.com/>.

Shortly, we'll be implementing a simple application to calculate Fibonacci numbers using Express, and in later chapters we'll be doing quite a bit more with Express. We'll also be exploring how to mitigate the performance problems from the computationally intensive code we discussed earlier.

Before we start implementing the Fibonacci application, we must set up and familiarize ourselves with Express.

Install Express globally in order to have access to the Express command-line tool.

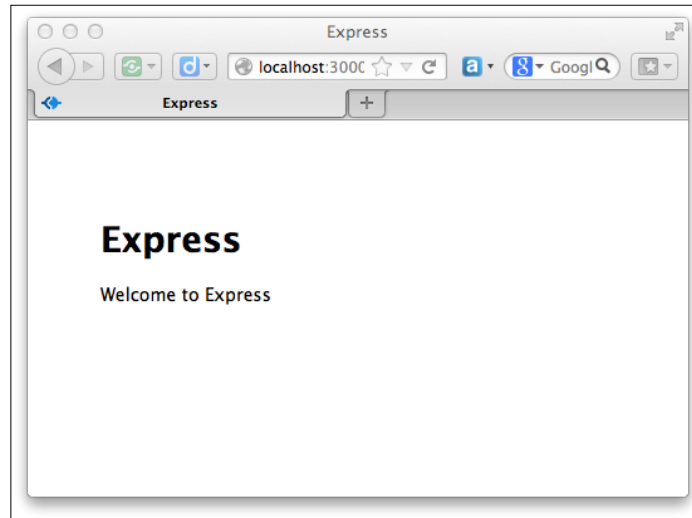
```
$ sudo npm install -g express
.. much output
/opt/local/bin/express -> /opt/local/lib/node_modules/express/bin/express
```

```
express@3.1.0 /opt/local/lib/node_modules/express
├─ methods@0.0.1
├─ fresh@0.1.0
├─ cookie-signature@0.0.1
├─ range-parser@0.0.4
├─ buffer-crc32@0.1.1
├─ cookie@0.0.5
├─ debug@0.7.0
├─ commander@0.6.1
├─ mkdirp@0.3.3
├─ send@0.1.0 (mime@1.2.6)
└─ connect@2.7.2 (pause@0.0.1, bytes@0.1.0, formidable@1.0.11, qs@0.5.1)
```

Now, set up a directory to create the Fibonacci application:

```
$ express --ejs fibonacci
  create : fibonacci
  create : fibonacci/package.json
  create : fibonacci/app.js
  create : fibonacci/public
  create : fibonacci/public/javascripts
  create : fibonacci/public/images
  create : fibonacci/public/stylesheets
  create : fibonacci/public/stylesheets/style.styl
  create : fibonacci/routes
  create : fibonacci/routes/index.js
  create : fibonacci/routes/user.js
  create : fibonacci/views
  create : fibonacci/views/index.ejs
install dependencies:
  $ cd fibonacci && npm install
run the app:
  $ node app
```

This created a blank Express application, contained in `app.js`. If you run the application using the suggested command line, it will print listening on port 3000. Visit that port in your browser (`http://localhost:3000`) and you'll be greeted by a default application screen:



The Express command-line tool is used solely to set up blank applications, and it takes several options to add various middleware to that initial application. You can add any of that middleware yourself, of course, but these options ensure these are set up the right way:

```
$ express --help
```

```
Usage: express [options]
```

```
Options:
```

<code>-h, --help</code>	output usage information
<code>-V, --version</code>	output the version number
<code>-s, --sessions</code>	add session support
<code>-e, --ejs</code>	add ejs engine support (defaults to jade)
<code>-J, --jshtml</code>	add jshtml engine support (defaults to jade)
<code>-H, --hogan</code>	add hogan.js engine support
<code>-c, --css <engine></code>	add stylesheet <engine> support (less stylus) (defaults to plain css)
<code>-f, --force</code>	force on non-empty directory

Walking through the default Express application

A quick read through the default application will show us quite a bit about using the Express framework to implement applications:

```
var express = require('express')
  , routes = require('./routes')
  , user = require('./routes/user')
  , http = require('http')
  , path = require('path');
```

This pulls in the required modules. Two of these modules, `routes` and `user`, come from the application inside the `routes` directory. As we'll see in a bit, they handle routing requests to request handler functions:

```
var app = express();
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));
app.configure('development', function() {
  app.use(express.errorHandler());
});
```

This section sets up and configures the Express application and middleware.

The `app.set` method provides settings for Express. Here, we set the port the server will be listening to, as well as the location for views and the default template engine to use. The `views` directory is used to store templates we'll use to render pages.

The `app.use` method configures middleware. In this case it sets up a favicon handler, request logging, a request body parser to handle form submissions, and a static file handler for the CSS, JavaScript, and image files.

The `app.configure` method supports conditional configuration depending on the environment variable `NODE_ENV`. The purpose is to allow for different behavior in different situations, such as adding extra debugging or testing hooks when run in a development environment.

```
app.get('/', routes.index);
app.get('/users', user.list);
http.createServer(app).listen(app.get('port'), function() {
  console.log("Express server listening on port " +
    app.get('port'));
});
```

The `app.get` function comes in two forms. The first simply retrieves an Express setting, such as the port on which to run the server. The other specifies the request handler function for a given route on an HTTP GET request.

The HTTP protocol uses a number of verbs like GET, PUT, and POST. The application object has a function like `app.get` for each of the verbs (`app.put`, `app.post`, and so on).

These functions are the routing functions in Express. They arrange for requests on a given URL, for a given HTTP verb, to be handled by a given handler function. This flexibility to handle any HTTP verb means an Express application could theoretically implement any type of HTTP interaction.

Calculating the Fibonacci sequence with Express

The Fibonacci numbers are the integer sequence: 0 1 1 2 3 5 8 13 21 34 ...

Each entry in the list is the sum of the previous two entries in the list, and the sequence was invented in 1202 by Leonardo of Pisa who was also known as Fibonacci. One method to calculate entries in the Fibonacci sequence is the recursive algorithm we showed earlier. We're going to create an Express application that uses the Fibonacci implementation and then explore several methods to mitigate performance problems in computationally intensive algorithms.

Let's start with the blank application we created in the previous step. We had you name that application `fibonacci` for a reason. We were thinking ahead.

In `app.js` make the following changes:

- Change `require('./routes/user')` to `fibonacci = require('./routes/fibonacci')`
- Delete the route `app.get('/users' ..` and in its place `app.get('/fibonacci', fibonacci.index);`

For the Fibonacci application we don't need to support users, but we need a page to query for a number for which we'll calculate the Fibonacci number. The `fibonacci` module we'll show in a minute will serve that purpose.

In the top level directory, create a file, `math.js`, containing this extremely simple Fibonacci implementation:

```
var fibonacci = exports.fibonacci = function(n) {
  if (n === 1)
    return 1;
  else if (n === 2)
    return 1;
  else
    return fibonacci(n-1) + fibonacci(n-2);
}
```

In the `views` directory, create a file named `top.ejs`:

```
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <h1><%= title %></h1>
  <div class='navbar'>
    <p><a href='/'>home</a>
    | <a href='/fibonacci'>Fibonacci's</a></p>
  </div>
```

This file contains the top part of the HTML pages we'll send to the users.

And another file named `bottom.ejs` contains:

```
</body>
</html>
```

This file contains the bottom part of the HTML pages.

Change `views/index.ejs` to just contain the following:

```
<% include top %>
<p>Welcome to the Math calculator</p>
<% include bottom %>
```

This serves as the front page of our application. While it doesn't contain any functionality, notice that `top.ejs` has a link to `/fibonacci`, which we'll look at next.

Together, `top.ejs` and `bottom.ejs` let us use a consistent look and overall page layout without having to replicate it in every page.

Create a file, `views/fibonacci.ejs`, containing the following code:

```
<% include top %>
<% if (typeof fiboval !== "undefined") { %>
  <p>Fibonacci for <%= fibonum %> is <%= fiboval %></p>
  <hr/>
<% } %>
<p>Enter a number to see its' Fibonacci number</p>
<form name='fibonacci' action='/fibonacci' method='get'>
  <input type='text' name='fibonum' />
  <input type='submit' value='Submit' />
</form>
<% include bottom %>
```

Remember that the files in `views` are templates into which data is rendered. They serve the View aspect of the Model-View-Controller paradigm, hence the directory name.

Each of the two views, `views/index.ejs` and `views/fibonacci.ejs`, are truncated pieces of a full HTML page. Where does the rest come from? From `top.ejs` and `bottom.ejs`, which are included by the other templates.

In the `routes` directory, delete the `user.js` module. It is generated by the Express framework, but we're not going to use it in this application.

Change `routes/index.js` to contain the following code:

```
exports.index = function(req, res) {
  res.render('index', {
    title: "Math Calculator"
  });
}
```

Then finally, in the `routes` directory, create a file named `fibonacci.js` containing the following code:

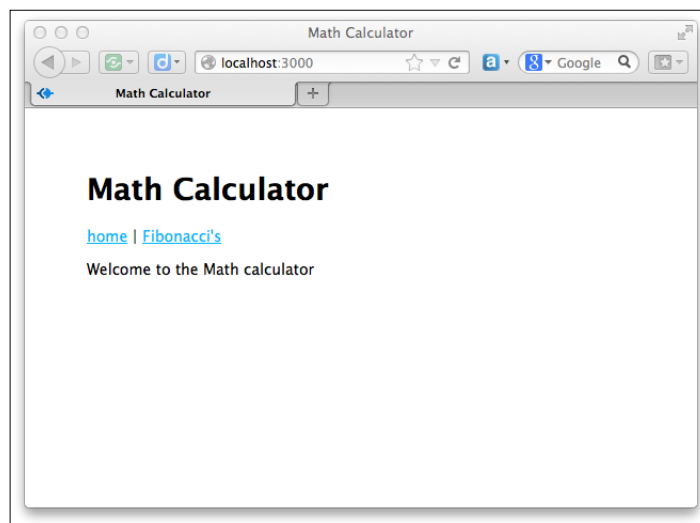
```
var math = require('../math');
exports.index = function(req, res) {
  if (req.query.fibonum) {
    res.render('fibonacci', {
      title: "Calculate Fibonacci numbers",
      fibonum: req.query.fibonum,
      fiboval: math.fibonacci(req.query.fibonum)
    });
  } else {
    res.render('fibonacci', {
      title: "Calculate Fibonacci numbers",
      fiboval: undefined
    });
  }
};
```

Now run the application:

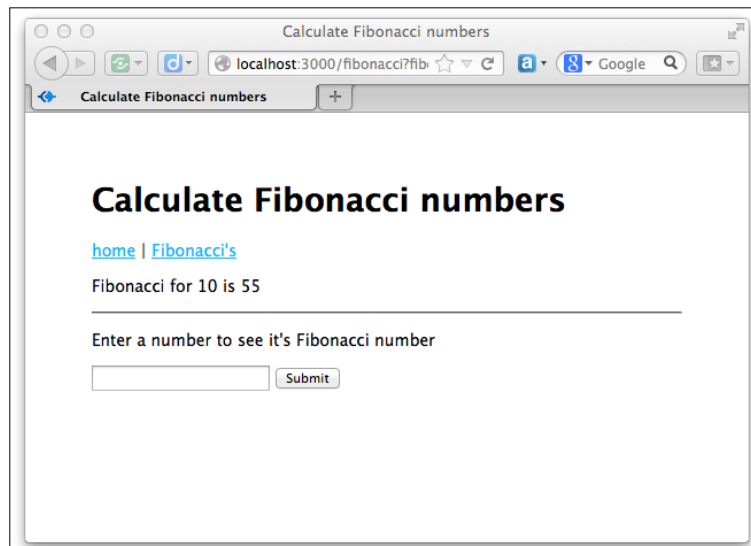
```
$ node app
```

```
Express server listening on port 3000
```

And then in your browser visit `http://localhost:3000` to see the application:



This page is rendered from the `views/index.ejs` template. Simply click on the Fibonacci link to go to the next page, which is of course rendered from the `views/fibonacci.ejs` template. On that page you'll be able to enter a number, click on the **Submit** button, and get an answer (hint: pick a number below 40 if you want your answer in a reasonable amount of time):



Let's walk through the application to discuss how it works.

There are two routes in `app.js`, the route for `/`, which is handled by `routes/index.js`, and the route for `/fibonacci`, which is handled by `routes/fibonacci.js`.

The `res.render` function renders the named template, using the provided data values, and emits the result as the HTTP response. For the home page of this application, the rendering code (`routes/index.js`) and template (`views/index.ejs`) aren't much, and it is on the Fibonacci page where all the action is happening.

The `views/fibonacci.ejs` template contains a form in which the user enters a number. Because it is a GET form, when the user clicks on the **Submit** button, the browser will issue an HTTP GET on the `/fibonacci` URL. What distinguishes one GET on `/fibonacci` from another is whether the URL contains a query parameter named **fibonum**. When the user first enters the page, there is no `fibonum` and hence nothing to calculate. After the user has entered a number and clicked on **Submit**, there is a `fibonum`, and something to calculate.

Express automatically parses the query parameters making them available as `req.query`. That means `routes/fibonacci.js` can quickly check if there is a `fibonum`. If there is, it calls the `fibonacci` function to calculate the value.

Earlier we asked you to enter a number less than 40. Go ahead. Enter a larger number, like 50, but go take a coffee break because this is going to take a while to calculate.

Computationally intensive code and the event loop

This Fibonacci example is purposely inefficient to demonstrate an important consideration for your applications. What happens to the Node event loop when running long computations? To see the effect, open two browser windows, each visiting the Fibonacci page. In one enter 55 or bigger, and in the other enter 10. Notice that the second window freezes and if you leave it running long enough (overnight) the answer will eventually pop up in both windows. What's happening is the Node event loop is being blocked from processing events, because the Fibonacci algorithm is running and does not ever yield to the event loop.

Since Node has a single execution thread, processing requests depends on request handlers quickly returning to the event loop. Normally, the asynchronous coding style ensures that the event loop executes regularly. This is true even for requests that load data from a server half way around the globe because I/O is non-blocking and control is quickly returned to the event loop. The naïve Fibonacci function we chose doesn't fit into this model, because it's a long running, blocking operation. This type of event handler prevents the system from processing requests and stops Node from doing what it's meant to do, namely to be a blistering fast web server.

In this case, the long-response-time problem is obvious. Response time quickly escalates to the point you can take a vacation to Tibet during the time it takes to respond with the Fibonacci number! Long response times might not be obvious in your application, so how do you know your requests are taking too long? One measurement to make is response latency shown by browser tools such as YSlow (<http://developer.yahoo.com/yslow/>). The rule of thumb when there are human beings using a web browser, is to show the next page within a second or two or else run the risk of losing your visitor.

To see this more clearly, create a file named `fibotimes.js` containing the following code:

```
var math = require('./math');
var util = require('util');
for (var num = 1; num < 80; num++) {
  util.log('Fibonacci for ' + num + ' = ' + math.fibonacci(num));
}
```

Then run it. You will get the following output:

```
$ node fibotimes
2 Feb 14:11:04 - Fibonacci for 1 = 1
2 Feb 14:11:04 - Fibonacci for 2 = 1
2 Feb 14:11:04 - Fibonacci for 3 = 2
2 Feb 14:11:04 - Fibonacci for 4 = 3
2 Feb 14:11:04 - Fibonacci for 5 = 5
2 Feb 14:11:04 - Fibonacci for 6 = 8
2 Feb 14:11:04 - Fibonacci for 7 = 13
2 Feb 14:11:04 - Fibonacci for 8 = 21
2 Feb 14:11:04 - Fibonacci for 9 = 34
..
2 Feb 14:11:52 - Fibonacci for 45 = 1134903170
2 Feb 14:12:21 - Fibonacci for 46 = 1836311903
2 Feb 14:13:08 - Fibonacci for 47 = 2971215073
```

This quickly calculates the first 40 or so members of the Fibonacci sequence, but after the 40th member it starts taking a couple seconds per result and quickly degrades from there. As you can see here, it took almost a minute to calculate the 47th member of the sequence. It is untenable to execute code of this sort on a single threaded system that relies on a quick return to the event loop.

There are two general ways in Node to solve this problem:

- **Algorithmic refactoring:** Perhaps, like the Fibonacci function we chose, one of your algorithms is suboptimal and can be rewritten to be faster. Or, if not faster, it can be split into callbacks dispatched through the event loop. We'll look at one such method in a moment.
- **Creating a backend service:** Can you imagine a backend server dedicated to calculating Fibonacci numbers? Okay, maybe not, but it's quite common to implement backend servers to offload work from frontend servers, and we will implement a backend mathematics server at the end of this chapter. The request handler should be making asynchronous calls to data services or databases, assembling everything required for the response, sending it to the browser when ready.

Algorithmic refactoring

To prove that we have an artificial problem on our hands, here is a much more efficient Fibonacci function:

```
var fibonacciLoop = exports.fibonacciLoop = function(n) {
  var fibos = [];
  fibos[0] = 0;
  fibos[1] = 1;
  fibos[2] = 1;
  for (var i = 3; i <= n; i++) {
    fibos[i] = fibos[i-2] + fibos[i-1];
  }
  return fibos[n];
}
```

If we substitute a call to `math.fibonacciLoop` in place of `math.fibonacci`, these programs run much faster. Even this isn't the most efficient implementation; for example, a simple prewired lookup table is much faster at the cost of some memory.

Some algorithms aren't so simple to optimize and still take a long time to calculate the result. However, it's possible to divide the calculation into chunks and then dispatch computation of those chunks through the event loop. Consider the following code:

```
var fibonacciAsync = exports.fibonacciAsync = function(n, done) {
  if (n === 0)
    done(undefined, 0);
  else if (n === 1 || n === 2)
    done(undefined, 1);
  else {
    setImmediate(function() {
      fibonacciAsync(n-1, function(err, val1) {
        if (err) done(err);
        else setImmediate(function() {
          fibonacciAsync(n-2, function(err, val2) {
            if (err) done(err);
            else done(undefined, val1+val2);
          });
        });
      });
    });
  }
}
```

This converts the `fibonacci` function from a synchronous function to an asynchronous function, one with a callback. By using `setImmediate`, each stage of the calculation is managed through Node's event loop, and the server can easily handle other requests while churning away on a calculation. It does nothing to reduce the computation required; it simply spreads that computation through the event loop.

Prior to Node 0.10.x, this asynchronous Fibonacci function was implemented using `process.nextTick` rather than `setImmediate`. For 0.10.x, Node detects recursive uses of `process.nextTick` and prints a message asking to use `setImmediate` instead.

This is not the only method to dispatch steps of an algorithm through the event loop. The `async` module can do this, and has a long list of functions that help tame asynchronous JavaScript.

Because it's an asynchronous function, we will need to change our application code in `routers/fibonacci.js` to the following:

```
var math = require('../math');
exports.index = function(req, res) {
  if (req.query.fibonum) {
    math.fibonacciAsync(req.query.fibonum, function(err, fiboval){
      res.render('fibonacci', {
        title: "Calculate Fibonacci numbers",
        fibonum: req.query.fibonum,
        fiboval: fiboval
      });
    });
  } else {
    res.render('fibonacci', {
      title: "Calculate Fibonacci numbers",
      fiboval: undefined
    });
  }
};
```

With this change, the server no longer freezes when calculating a large Fibonacci number. The calculation of course still takes a long time, but at least other users of the application aren't blocked.

You can verify this by again opening two browser windows on the application, entering 55 in one window and in the other start requesting smaller Fibonacci numbers.

It's up to you, and your specific algorithms, to choose how to best optimize your code and to handle any long running computations you may have.

Making HTTP Client requests

The next way to mitigate computationally intensive code is to push the calculation to a backend process. To do that we'll request computations from a backend Fibonacci server, using the HTTP Client object to do so. But before we look at that, let's first talk in general about using the HTTP Client object.

Node includes an HTTP Client object useful for making HTTP requests. It has enough capability to issue any kind of HTTP request, but for example it does not emulate a full browser, so don't get delusions of this being a full-scale test automation tool. Its scope focuses solely on the HTTP protocol. It's possible to build a browser emulator on top of this HTTP client, for example to build a test automation tool. The HTTP Client object can be used for any kind of HTTP request, such as calling a **Representational State Transfer (REST)** web service.

Let's start with some code inspired by the `wget` or `curl` commands to make HTTP requests and show the results. Create a file named `wget.js` containing this code:

```
var http = require('http');
var url = require('url');
var util = require('util');
var argUrl = process.argv[2];
var parsedUrl = url.parse(argUrl, true);
var options = {
  host: null,
  port: -1,
  path: null,
  method: 'GET'
};
options.host = parsedUrl.hostname;
options.port = parsedUrl.port;
options.path = parsedUrl.pathname;
if (parsedUrl.search) options.path += "?" + parsedUrl.search;
var req = http.request(options, function(res) {
  util.log('STATUS: ' + res.statusCode);
  util.log('HEADERS: ' + util.inspect(res.headers));
  res.setEncoding('utf8');
  res.on('data', function(chunk) {
    util.log('BODY: ' + chunk);
  });
  res.on('error', function(err) {
    util.log('RESPONSE ERROR: ' + err);
  });
});
req.on('error', function(err) {
```

```
    util.log('REQUEST ERROR: ' + err);
  });
  req.end();
```

You can run the script as follows:

```
$ node wget.js http://example.com
11 Apr 21:34:35 - STATUS: 302
11 Apr 21:34:35 - HEADERS: { location: 'http://example.iana.org',
  server: 'BigIP',
  connection: 'Keep-Alive',
  'content-length': '0' }
```

This shows an HTTP response with status code 302 (redirect) telling your browser to instead go to `http://example.iana.org/`, and indeed if you visit `http://example.com` in your browser it will redirect over to the `iana.org` page.

The purpose of `wget.js` is to make an HTTP request and show you the voluminous detail of the response.

An HTTP request is initiated with the `http.request` method as follows:

```
var http = require('http');
var options = {
  host: 'example.com',
  port: 80,
  path: null,
  method: 'GET'
};
var request = http.request(options,
  function(response) { .. });
```

The options object describes the request to make, and the callback function is called when the response arrives. The options object is fairly straightforward with the host, port, and path fields specifying the URL being requested. The method field must be one of the HTTP verbs (GET, PUT, POST, and so on). You can also give a headers array for the headers in the HTTP request.

For example, you might need to provide a cookie:

```
var options = {
  headers: {
    'Cookie': '.. cookie value'
  }
};
```

The response object is itself an `EventEmitter`, which emits the data and error events. The data event is called as data arrives, and the error event is of course called on errors.

The request object is a `WritableStream` (http://nodejs.org/api/stream.html#stream_class_stream_writable), which is useful for HTTP requests containing data, such as PUT or POST. This means the request object has a `write` function that writes data to the requester. The data format in an HTTP request is specified by the MIME standard originally created to give us better e-mail. Around 1992 the WWW community worked with the MIME standard committee to apply portions of MIME for use in HTTP. HTML forms will POST with a Content-Type of `multipart/form-data`, for example.

Calling a REST backend service from an Express application

Now that we've seen how to make HTTP client requests, we can look at how to make a REST query inside an Express web application. What that effectively means is to make an HTTP GET request to a backend server, which responds with the Fibonacci number represented by the URL. To do so we'll be refactoring the Fibonacci application to make a Fibonacci server that is called from the application. While this is overkill for calculating Fibonacci numbers, it lets us look at the basics of implementing a multi-tier application stack.

Inherently calling a REST service is an asynchronous operation. That means calling the REST service will involve a function call to initiate the request, and a callback function to receive the response. REST services are accessed over HTTP, so we'll be using the HTTP client object to do so. The general pattern for an Express request handler in this case is as follows:

```
exports.index = function(req, res) {
  callAsyncOrRESTservice(function(err, data) {
    res.render(template, {
      // data for the template
    });
  });
}
```


Implementing a simple REST server with Express

While Express has a powerful templating system making it suitable for delivering HTML web pages to browsers, it can also be used to implement a simple REST service. The Express router supports parameterized URLs that lets Express treat parts of the URL as if they were function parameters.

Parameterized URLs can make your program more flexible than non-parameterized URLs. It's done by a kind of pattern matching with tokens plugged into URL elements. Express examines the request URL, matching it against the patterns you specify, extracting matching elements from the URL, and filling the data into fields in the `req` object.

An example might make this clearer:

```
app.get('/user/:id', function(req, res){
  res.send('user ' + req.params.id);
});
```

In the URL, `/user/:id` has a placeholder token named `id`. Express recognizes the stuff after `/user/` and assigns it to the `req.params.id` field. The pattern can be a regular expression, if you prefer.

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

```
var math = require('./math');
var express = require('express');
var util = require('util');
var app = express();
app.configure(function(){
  app.use(app.router);
});
app.get('/fibonacci/:n', function(req, res, next) {
  math.fibonacciAsync(Math.floor(req.params.n),
    function(err, val) {
      if (err) next('FIBO SERVER ERROR ' + e);
      else {
        res.send({
          n: req.params.n,
          result: val
        });
      }
    }
  );
});
app.listen(3002);
```

This is a stripped down Express application, that gets right to the point of providing a Fibonacci calculation service. The one route it supports handles the Fibonacci computation using the same functions we've already worked with.

This is the first time we've seen `res.send` used. It's a flexible way to send responses which can take an array of header values (for the HTTP response header), and an HTTP status code. As used here, it automatically detects the object, formats it as JSON text, and sends it with the correct Content-Type.

Now let's run it:

```
$ node fiboserver.js
```

Then, in a separate command window, run these commands. If your computer does not have the `curl` command, skip this step and use the `fiboclient.js` script instead.

```
$ curl -f http://localhost:3002/fibonacci/10
{ "n": "10", "result": 55 }
$ curl -f http://localhost:3002/fibonacci/11
{ "n": "11", "result": 89 }
```

Now create a simple client program for this server in `fiboclient.js`:

```
var http = require('http');
var util = require('util');
[
  "/fibonacci/30", "/fibonacci/20", "/fibonacci/10",
  "/fibonacci/9", "/fibonacci/8", "/fibonacci/7",
  "/fibonacci/6", "/fibonacci/5", "/fibonacci/4"
].forEach(function(path) {
  var req = http.request({
    host: "localhost",
    port: 3002,
    path: path,
    method: 'GET'
  }, function(res) {
    res.on('data', function (chunk) {
      util.log('BODY: ' + chunk);
    });
  });
  req.end();
});
```

At the top we have an array of URLs with which we will make requests. This simply uses `http.request`, formats the correct URL, makes the request, and prints out the result.

Now let's run it:

```
$ node fiboclient.js
2 Feb 16:11:48 - BODY: { "n": "10", "result": 55 }
2 Feb 16:11:48 - BODY: { "n": "9", "result": 34 }
2 Feb 16:11:48 - BODY: { "n": "8", "result": 21 }
2 Feb 16:11:48 - BODY: { "n": "5", "result": 5 }
2 Feb 16:11:48 - BODY: { "n": "7", "result": 13 }
2 Feb 16:11:48 - BODY: { "n": "6", "result": 8 }
2 Feb 16:11:48 - BODY: { "n": "4", "result": 3 }
2 Feb 16:11:48 - BODY: { "n": "20", "result": 6765 }
2 Feb 16:11:50 - BODY: { "n": "30", "result": 832040 }
```

Notice that the results arrived out-of-order from when the requests were made. The order of response is likely to differ for you, because they are asynchronous operations. The results will arrive when they're ready and not a millisecond sooner. We've already shown that larger Fibonacci values take longer to compute. We shouldn't be surprised, then, to see that the result for the 30th Fibonacci number arrived last, even though it was requested first, because it takes so much longer to calculate than the smaller Fibonacci values.

What happened is that `fiboclient.js` sends all its requests right away, and then each one goes into a wait for the response to arrive. Because the server is using `fibonacciAsync`, it will work on calculating all responses simultaneously. The values that are quickest to calculate are the ones that will be ready first. As the responses arrive in the client, the matching response handler fires, and in this case the result is printed to the console.

Refactoring the Fibonacci application for REST

Now that we've looked into implementing a REST-based server, we can return to the Fibonacci application, applying what we've learned to improve it. We will lift some of the code from `fiboclient.js` and transplant it into the application to do this. Change `routes/fibonacci.js` to the following code:

```
var math = require('../math');
exports.index = function(req, res) {
```

```
if (req.query.fibonum) {
  var httpreq = require('http').request({
    host: "localhost",
    port: 3002,
    path: "/fibonacci/"+Math.floor(req.query.fibonum),
    method: 'GET'
  }, function(httpresp) {
    httpresp.on('data', function (chunk) {
      var data = JSON.parse(chunk);
      res.render('fibonacci', {
        title: "Calculate Fibonacci numbers",
        fibonum: req.query.fibonum,
        fiboval: data.result
      });
    });
  });
  httpreq.end();
} else {
  res.render('index', {
    title: "Calculate Fibonacci numbers",
    fiboval: undefined
  });
}
};
```

Getting the whole system running will now take one more step. We first run `fiboserver` to make sure the REST service is running. Then we run `app.js`.

Because we haven't changed the templates, the screen will look exactly as it did earlier.

Some RESTful Node modules

Here are a few Node modules that make it easier to implement REST and HTTP Client code:

- <https://github.com/coolaj86/abstract-http-request>: Higher level wrapper around the HTTP request system
- <http://github.com/mikeal/request>: Simplified HTTP request client
- <https://github.com/danwrong/restler>: A REST client library
- <https://github.com/cloudhead/http-console>: A useful interactive shell for HTTP requests
- <http://percolatorjs.com>: Percolator.JS is a framework for developing well-designed JSON based APIs

Summary

We learned a lot in this chapter about Node's HTTP support, and implementing web applications. In particular, we learned about the following topics:

- EventEmitter
- Listening to HTTP events
- A simple web application with no frameworks
- The Express application framework
- How to deal with computationally intensive code
- Constructing a multi-tier application with a REST service

Now we can move on to implementing a more complete application, one for taking notes. We will use the Notes application for several upcoming chapters as a vehicle to explore the Express application framework, database access, deployment to cloud services or on your own server, and user authentication. In the next chapter, we will build the basic infrastructure of a simple Express application.

5

Implementing a Simple Express Application

Now that we've got our feet wet building an Express application for Node, let's work on an application that performs a useful function. The application we'll build will keep a list of notes, and will let us explore some aspects of a real application. In this chapter, we'll only build the basic infrastructure of the application, and in the later chapters we'll add features to it, such as using different database engines to store the notes, user authentication, deployment on public servers, and other things.

Express and the MVC paradigm

Express doesn't follow a pure **model-view-controller** (MVC) structure, but the blank application created by the `express` command line tool provides you with two MVC aspects. These are stated as follows:

- The `views` directory contains template files, hence controls the data display, and corresponds to the view
- The `routes` directory handles requests on URLs, hence sends commands inside the application, and corresponds to the controller.

This leaves one wondering how to handle code implementing the model. The role of the model is to hold the application data, changing the data as instructed by the controller, and supplying the data requested by the view.

At a minimum, the model code should be in a separate module from the controller code. That's because, ideally, the model and controller code should be as independent from each other as possible.

The approach we'll use in developing the Notes application is to create a `models` directory as a sibling of `views` and `routes`. The `models` directory will hold modules for storing the notes and related data. The API of these modules will provide functions to create, read, update, or delete data items (the CRUD model) and other functions necessary for the view to do its thing.

The **CRUD** model stands for **Create**, **Read**, **Update**, and **Delete** (or **Destroy**). These are the four basic operations of persistent storage. The Notes application is structured as a CRUD application, to demonstrate implementing each of these basic operations.

We'll use functions named `create`, `read`, `update`, and `destroy` to implement each of the basic operations. We're using the verb `destroy` rather than `delete`, because `delete` is a reserved word in JavaScript.

Creating the Notes application code

Let's start, as before, with the `express` command line tool:

```
$ express --ejs notes
.. instructions
$ cd notes
$ npm install
.. output
$ ls
app.js node_modules package.json public routes views
```

If you wish, you can run `node app` to view the blank application in your browser. But instead, let's move on to setting up the code.

The Notes model

Create a directory named `models`, as a sibling of the `views` and `routes` directories. Then create a file named `notes.js` in that directory, with this code:

```
var notes = [];
exports.update = exports.create = function(key, title, body) {
  notes[key] = { title: title, body: body };
}
exports.read = function(key) {
  return notes[key];
}
exports.destroy = function(key) {
  delete notes[key];
}
```

```
exports.keys = function() {  
  return Object.keys(notes);  
}
```

This is a simple in-memory data store that's fairly self-explanatory. It does not support any long-term data persistence. Any data stored in this model will disappear if the server is killed.

The `create` and `update` functions are being handled by the same function. At this stage of the Notes application, the code for both these functions can be exactly the same because they perform the exact same operation.

Later, when we add database support to Notes, the `create` and `update` functions will need to be different. For example, in a SQL data model, `create` would be implemented with an `INSERT INTO` command, while `update` would be implemented by an `UPDATE` command.

Each note is identified by a key, which is allowed to be any value JavaScript supports as an array index. As we'll see, the key is passed around, as an identifier in forms for creating or editing notes, and for retrieving notes from the data store.

The Notes home page

In `app.js` we already have this route, which uses `routes/index.js` to handle the home page:

```
app.get('/', routes.index);
```

Also, in `app.js`, add this `require` statement at the top:

```
, notes = require('./routes/notes')
```

Then remove the `require` statement for the `./routes/user` module, as well as the matching route that reads as follows:

```
app.get('/users', user.list);
```

Change `routes/index.js` to contain this code:

```
var notes = require('../models/notes');  
exports.index = function(req, res) {  
  res.render('index', { title: 'Notes', notes: notes });  
};
```

Notice that we access the Model code using a relative `require` statement, referring to the module we just created in the `models` directory.

We'll also provide view access to the Model so it can retrieve the list of notes.

Now change `views/index.ejs` to contain this code:

```
<% include top %>
<%
var keys = notes.keys();
if (keys) {
  keys.forEach(function(key) {
    var note = notes.read(key);
    %><p><%= key %>:
    <a href="/noteview?key=<%= key %>"><%= note.title %></a>
    </p><%
  });
} %>
<% include bottom %>
```

The view is given a reference to the `notes` module, so that it can request Notes data from the model. While writing a template, it helps, for long-term maintainability, if the template code does not know any implementation details. Ideally one could swap in a different model implementation, such as to use a database, and make no changes to view templates.

In this case, we're looping over all notes, and outputting a link to view the note. The link target, `/noteview`, is something we'll implement in a minute.

Now add a file, `views/top.ejs`, to contain the code for the top portion of each page:

```
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <h1><%= title %></h1>
  <div class='navbar'>
    <p><a href='/'>Home</a> | <a href='/noteadd'>ADD Note</a></p>
  </div>
```

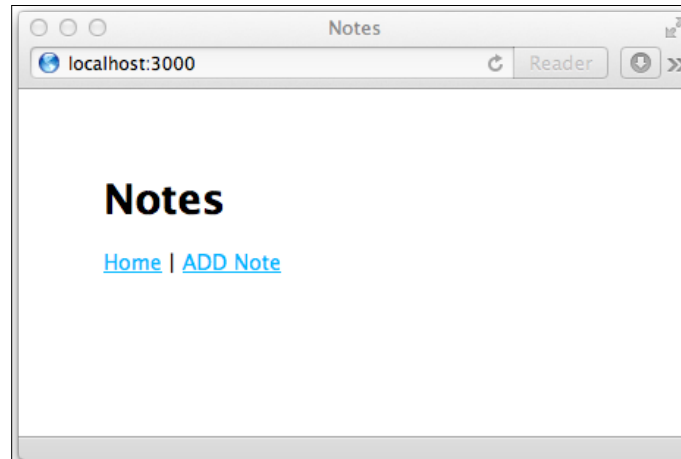
And add a file, `views/bottom.ejs`, to contain the code for the bottom portion:

```
</body>
</html>
```

You can run the application at this point:

```
$ node app
```

And you will see the following:



Because there aren't any notes (yet), the screen doesn't show any notes. Clicking on the **Home** link just refreshes this page, while clicking on **ADD Note** will give you an error message because we haven't written that code yet.

Adding a new note (create)

Now, let's look at how to add (create) a note.

Because the application doesn't have a route configured for the `/noteadd` URL, we must add one. We will do so by adding the following route to `app.js`:

```
app.get('/noteadd', notes.add);
```

This route matches the **ADD Note** link in the `top.ejs` template.

Next, create a file in `routes`, named `notes.js`, to contain the route functions for the create, read, update, and delete operations on Notes.

Start `notes.js` with this code:

```
var notes = require('../models/notes');
exports.add = function(req, res, next) {
  res.render('noteedit', {
    title: "Add a Note",
    docreate: true,
    notekey: "",
    note: undefined
  });
}
```

And in the `views` directory, add a file named `noteedit.ejs`, containing the following:

```
<% include top %>
<form method='POST' action='/notesave'>
<input type='hidden' name='dcreate'
      value='<%=dcreate ? "create" : "update"%>'>
<p>Key: <input type='text' name='notekey'
      value='<%= note ? notekey : "" %>'></p>
<p>Title: <input type='text' name='title'
      value='<%= note ? note.title : "" %>' /></p>
<br/><textarea rows=5 cols=40 name='body' ><%=
      note ? note.body : "" %></textarea>
<br/><input type='submit' value='Submit' />
</form>
<% include bottom %>
```

Why is the template named `noteedit.ejs` when we're adding a new note, not editing an existing one? It's because we're going to reuse the same template for both creating and editing notes.

Notice that the `note` and `notekey` objects passed to the template are empty. The template detects this condition and ensures that the input areas in the template are empty. Additionally a flag, `dcreate`, is passed in so that the form records whether it is being used to create or update a note. At this point we are adding a new note, so of course no `note` object exists. The template code is being written defensively so it won't throw errors if it doesn't have a `note`.

This template is a form which will `POST` its data to the `/notesave` URL. If you were to run the application at this time, it would give an error message, because no route is configured for the `/notesave` URL.

To support this functionality, add the following route in `app.js`:

```
app.post('/notesave', notes.save);
```

And add the following code in `routes/notes.js`:

```
exports.save = function(req, res, next) {
  if (req.body.dcreate === "create") {
    notes.create(req.body.notekey,
      req.body.title, req.body.body);
  } else {
    notes.update(req.body.notekey,
      req.body.title, req.body.body);
  }
  res.redirect('/noteview?key='+req.body.notekey);
}
```

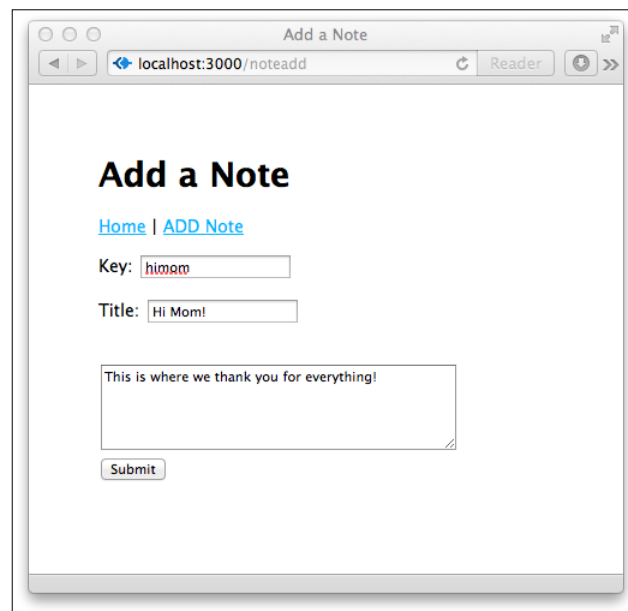
Because this code is called as the result of a `POST` operation, its data is added to the `req.body` object that's created by the `bodyParser` middleware. This middleware was added in `app.js` with this line of code:

```
app.use(express.bodyParser());
```

This `save` function is used for both adding a new note, and updating an existing one. A flag, `dcreate`, is passed around to inform the `save` function whether we are creating a new note or updating an existing one. The flag begins in the `add` function, where it is passed to the `noteedit.js` template. In that template, a form value named `dcreate` is created with either the value `create` or `update`. Then in the `save` function, this flag is checked, and either the `create` or `update` function is called.

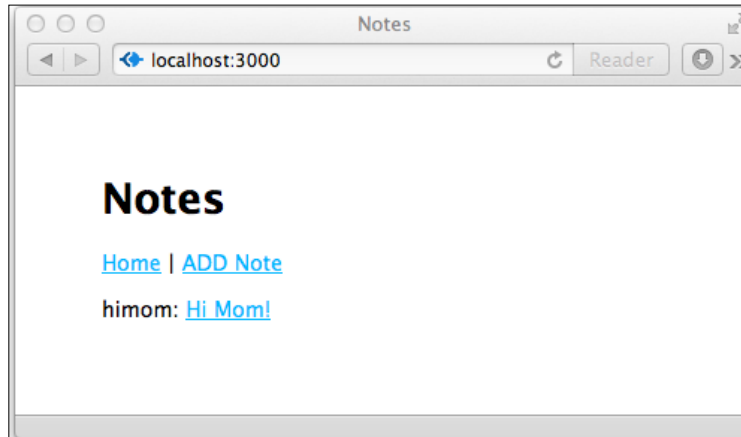
In the end, it doesn't matter with our current Notes model implementation because both the `notes.create` and `notes.update` functions are the same. However, as we noted earlier while discussing the model, we could change the implementation later so these functions perform different operations. Therefore, it will help us later to ensure the route function calls the appropriate model function based on the request.

Now we can run the application, and the **ADD Note** link will work.



But upon clicking the **Submit** button we get an error message because there isn't any code, yet, to implement the `/noteview` URL.

If you modify the URL to visit `http://localhost:3000`, you'll see the following in the home page:



Viewing notes (read)

Now that we've seen how to create a note, we must look at how to view one. Recall that in `views/index.ejs` we created a link to view notes this way:

```
<a href="/noteview?key=<%= key %>"><%= note.title %></a>
```

As we saw in the previous section, no route is configured for the `/noteview` URL.

To support this URL, add the following in `app.js`:

```
app.get('/noteview', notes.view);
```

And in `routes/notes.js` add this code:

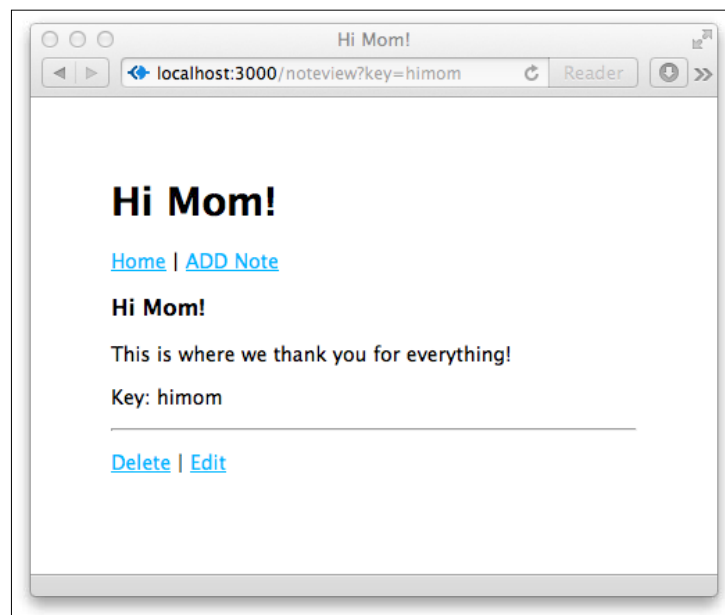
```
exports.view = function(req, res, next) {
  var note = undefined;
  if (req.query.key) {
    note = notes.read(req.query.key);
  }
  res.render('noteview', {
    title: note ? note.title : "",
    notekey: req.query.key,
    note: note
  });
}
```

And then in the `views` directory add the template, `noteview.ejs`, containing:

```
<% include top %>
<h3><%= note ? note.title : "" %></h3>
<p><%= note ? note.body : "" %></p>
<p>Key: <%= notekey %></p>
<% if (notekey) { %>
  <hr/>
  <p><a href="/notedestroy?key=<%= notekey %>">Delete</a>
    | <a href="/noteedit?key=<%= notekey %>">Edit</a></p>
<% } %>
<% include bottom %>
```

This code has a certain amount of defensive coding, in case the query parameter, `key`, was not provided, or otherwise no `note` object is provided. If the query parameter is missing, the `note` object will be undefined, and this template will simply be empty rather than displaying an error message.

Now we can run the application again, and see the following screen:



Notice that the template includes links to the `/notedestroy` and `/noteedit` URLs. But, if we click on either link, the browser will give an error message.

Editing an existing note (update)

Now that we've looked at the create and read operations, let's look at how to update or edit a note. In case it's not clear, we are implementing the four CRUD operations.

Again, no route is yet configured for the `/noteedit` URL. To support this, create this route in `app.js`:

```
app.get('/noteedit', notes.edit);
```

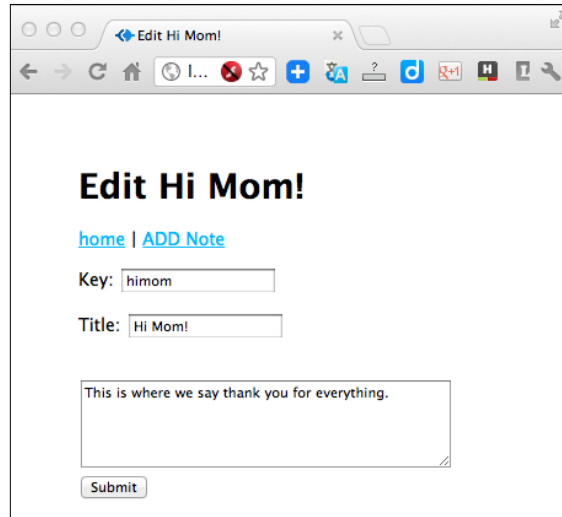
Then in `routes/notes.js`, add this function:

```
exports.edit = function(req, res, next) {
  var note = undefined;
  if (req.query.key) {
    note = notes.read(req.query.key);
  }
  res.render('noteedit', {
    title: note ? ("Edit " + note.title) : "Add a Note",
    docreate: note ? false : true,
    notekey: req.query.key,
    note: note
  });
}
```

This is of course reusing the `noteedit.ejs` template. Unlike for the `/noteadd` route function, in this case we have retrieved a `note` object and have passed it to the template. This way the template is set up to update an existing note rather than create a new one. That is, supply a reference to the `note` object, and pass `false` in the `docreate` field. With `docreate` set to `false`, the `noteedit.ejs` template then notifies the `save` function that it is updating an existing note.

What if the `/noteedit` URL is requested but no key is specified? In this case, the view is configured, like we did for the `/noteadd` route function, to act the same way, so that even though it's the `edit` function which was called, it will behave as if the `add` function was called instead.

Now upon running the application we can see the following screen; however, because we're using a non-persistent data model, you'll have to re-enter this text:



Deleting notes (destroy)

Now let's look at how to handle the `/notedestroy` route, to delete notes.

In `app.js` add the following route:

```
app.get('/notedestroy', notes.destroy);
```

Then add this function to `routes/notes.js`:

```
exports.destroy = function(req, res, next) {
  var note = undefined;
  if (req.query.key) {
    note = notes.read(req.query.key);
  }
  res.render('notedestroy', {
    title: note ? note.title : "",
    notekey: req.query.key,
    note: note
  });
}
```

And, in the `views` directory, add a file named `notedestroy.ejs`:

```
<% include top %>
<form method='POST' action='/notedestroy'>
<input type='hidden' name='notekey'
```



```
value='<%= note ? notekey : " " %>'  
<p>Delete <%= note.title %> ?</p>  
<br/><input type='submit' value='DELETE' />  
    <a href="/noteview?key=<%= notekey %>">Cancel</a>  
</form>  
<% include bottom %>
```

This is a form which will POST to the `/notedodestroy` URL. If you decide against deleting the note, there is a convenient link, **Cancel**, to bring you back to the `/noteview` screen.

If you were to run the Notes application right now, clicking on the **Delete** button will print an error message because no route is configured for the `/notedodestroy` URL.

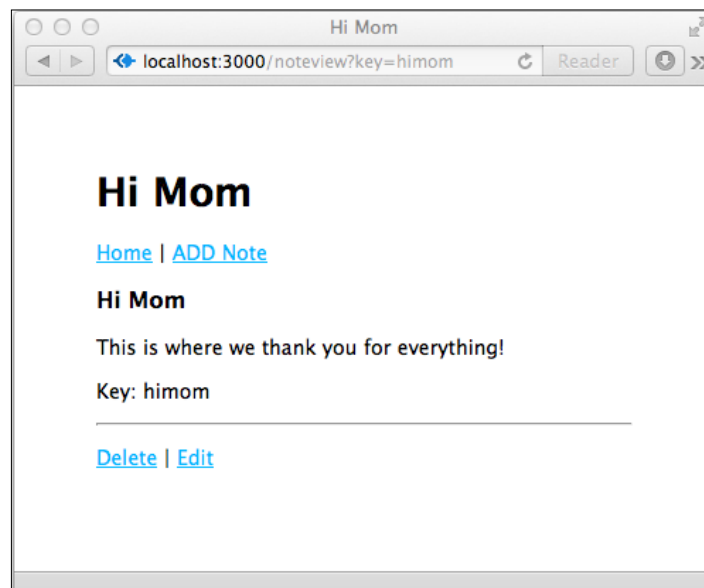
To handle `/notedodestroy`, add this route in `app.js`:

```
app.post('/notedodestroy', notes.dodestroy);
```

And then in `routes/notes.js` add this function:

```
exports.dodelete = function(req, res, next) {  
  notes.delete(req.body.notekey);  
  res.redirect('/');  
}
```

And now you can run the application again, and see the following screen:



Once you click on the **Delete** button, you'll go back to the Notes application home page after having deleted the note. If, on the other hand, you click on the **Cancel** link, you'll instead return to viewing the note.

Changing the look of an Express application

While the default Express theme is fairly nice, we'll always want to tweak how the application looks to suit our own preferences. That way it's our application.

Express provided us with this line of code in the `<head>` section of the HTML:

```
<link rel='stylesheet' href='/stylesheets/style.css' />
```

This CSS file is stored in the `public/stylesheets` directory. The `public` directory has three child directories; `images`, `javascripts`, and `stylesheets`. These directories represent the most common static assets you could use in an application.

Additionally, the whole `public` directory is served by the Express framework as if it were a regular web server. In `app.js` this is configured as follows:

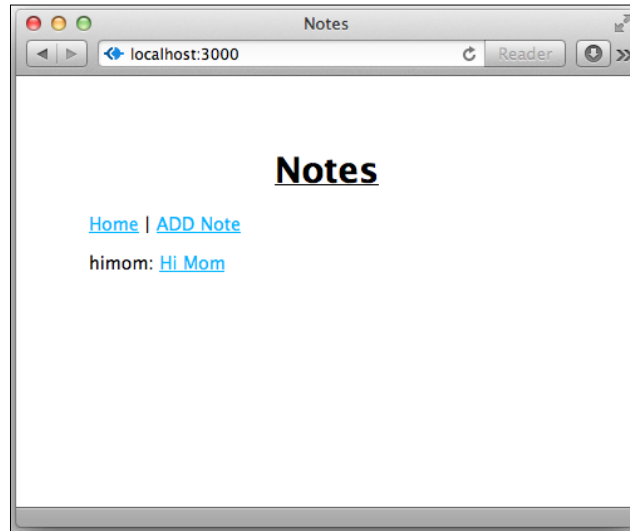
```
app.use(express.static(path.join(__dirname, 'public')));
```

Because any file you drop into the `public` directory is served to web browsers, you must ensure that you do not configure an application route that conflicts with one of the static assets. It also means that you can add files in the `public` directory, such as images, and CSS or JavaScript files, that customize the application's look and behavior.

For simple theming changes you can simply add CSS directives to `stylesheets/style.css`, like this:

```
h1 {  
  text-align: center;  
  text-decoration: underline;  
}
```

This changes all `<h1>` tags to be centered and underlined. Rerun the Notes application; you will see that it looks as follows:



It's possible to pull in other CSS files and make more extensive theming changes, such as making a more attractive look to the application, simply by modifying the `top.ejs` template.

For a more massive change you could install a framework such as Twitter Bootstrap. It is CSS and JavaScript that Twitter Inc. makes available for others to use. Bootstrap provides excellent defaults for compatibility with a broad range of browsers, a fluid grid system, a large set of useful components, typography enhancements, and more.

You can read up on it at <http://twitter.github.io/bootstrap/>, and it is easy to install in your application.

Download the ZIP file from the website and unpack it into the Notes application's public directory.

Next, in `views/top.ejs` add these lines within the `<head>` section of the template:

```
<meta name="viewport"
      content="width=device-width, initial-scale=1.0">
<link href="/bootstrap/css/bootstrap.min.css"
      rel="stylesheet" media="screen">
<link rel='stylesheet' href='/stylesheets/style.css' />
```

Including our own CSS file after including the Bootstrap CSS means we can easily override the Bootstrap CSS.

And add this at the very beginning of `views/top.ejs` to ensure pages are understood as being HTML5:

```
<!DOCTYPE html>
```

Then in `views/bottom.ejs`, add these two lines just before the closing `body` and `html` tags:

```
<script src="http://code.jquery.com/jquery.js"></script>
<script src="/bootstrap/js/bootstrap.min.js"></script>
</body></html>
```

Rerun the Notes application and you'll notice a few subtle changes. For example, the focus indicator on text entry fields on the Notes' create/edit page fades in and out.

Additionally, Bootstrap brings in the jQuery library making it possible to use any jQuery effect or plugin in your application.

Scaling up and running multiple instances

Now that we've gotten this far, surely you'll have played around with the application a bit, and will have created, read, updated, and deleted many notes.

Suppose for a moment that this isn't a toy application, but one that is interesting enough to draw a million users a day. Serving a high load typically means adding servers, load balancers, and many other things. A core part is to have multiple instances of the application running at the same time to spread out the load.

Let's see, then, what happens when you run multiple instances of the Notes application at the same time.

The first thing to consider is making sure the instances are on different ports. In the default version of `app.js` you will find this line:

```
app.set('port', process.env.PORT || 3000);
```

What this does is look in the user's environment variables, and if `PORT` is set, then the application will run on that port (`localhost:PORT`). If `PORT` is **not** set then it defaults to 3000, explaining why we've been using `http://localhost:3000` all this time.

To start two (or more) instances, each listening on different ports, each instance must be started while the `PORT` environment variable has a different value.

We can tell Notes to use a different port number by setting the environment variable appropriately:

```
$ PORT=3030 node app.js
Express server listening on port 3030
```

This is the way, on Unix, with the BASH shell, to set an environment variable when executing a command. For other environments one does this differently. For example, on Windows the equivalent command sequence is:

```
C:\> SET PORT=3030
C:\> node app.js
```

Then, to start the second instance, listening to a different port, open another terminal window and enter this command:

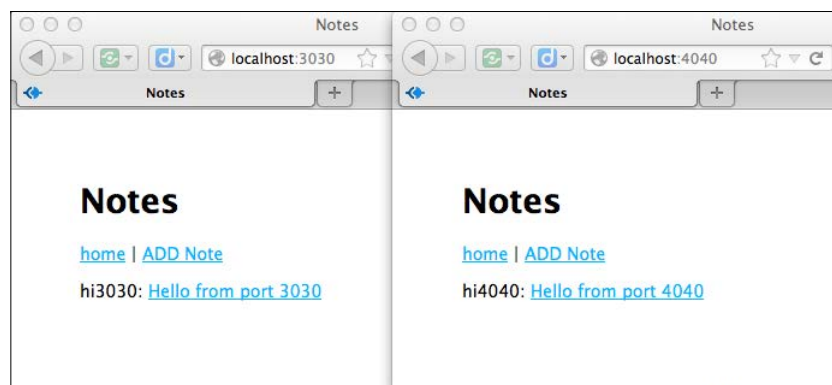
```
$ PORT=4040 node app.js
Express server listening on port 4040
```

Or on Windows, use the following:

```
C:\> SET PORT=4040
C:\> node app.js
```

Now you have two servers, one you can access at `http://localhost:3030`, and the other at `http://localhost:4040`.

After some editing and adding notes, your screen could look like this:



What's going on? Why is there different content shown in each browser window?

The reason for this is that each instance is running in its own instance of the Notes application. Each Notes instance has its own memory space, and there is no shared state or data between the two Notes instances, especially because our model code does not persist data in a database. Therefore each instance can only show the notes you have entered in that instance.

Depending on your needs, this behavior may, or may not, be desirable. Typically you run multiple instances to scale performance, and in that case, the data needs to be shared so that each instance accesses the same data. Typically this would be achieved by storing the notes in a database.

The primary purpose of a database is long-term reliable data storage, especially of complex data sets with complex queries. But a database also provides a way to share data when we run multiple instances.

In the next chapter, we will be looking at different database choices for Node, and add a database backend to the Notes application.

Summary

We've come quite a long way in this chapter.

- We built a useful working application with Express
- We implemented model-view-controller concepts in Express
- We implemented CRUD operations in Express
- We looked at handling form submissions in Express
- We looked at implementing the navigation structure of an application in Express
- We looked at how to change the look and feel of our application
- We looked at why a database is needed, not just for long-term data storage, but for sharing data when scaling up an application

In the next chapter, we'll be looking at several database engines.

6

Data Storage and Retrieval

In the previous chapter, we built a small and somewhat useful application for storing notes. While the application works reasonably well, it doesn't store those notes anywhere for long term storage. Further, if you run multiple instances of Notes each instance has its own set of notes, making it difficult to scale the application to serve lots of users.

The typical next step in such an application is to introduce a database tier. Databases provide long-term reliable storage, while enabling easy sharing of data between multiple application instances.

In this chapter we will be looking at database support in Node in order to solve these user stories:

- The user must see the same set of notes for any Notes instance accessed
- Reliably store notes for long term retrieval

We'll be starting with the Notes application code used in the previous chapter. We used a simple in-memory data model using an array to store the notes. In this chapter we'll add additional model implementations using several database engines.

Let's get started!

The first step will be to rename our existing model to match the other model implementations we'll work on in this chapter:

```
$ mv models models-memory
```

For the other models, the directory name will show the data storage engine being used.

Asynchronizing the Notes application

The previous Notes application used a synchronous model for accessing the data model. We got away with that because the data was stored in memory, and there was no need to access model data via asynchronous functions. However, by storing the data in a database, we're in territory where we must access the data via asynchronous functions. Recall the discussion in *Chapter 1, About Node*, about the original reasoning behind Node's asynchronous architecture. Accessing data in memory takes a few clock cycles making it possible to directly operate on in-memory data without delays that would prevent the Node server from handling events. Data stored anywhere else, even in the memory space of another process, requires a delay to access that data, and any delay prevents the Node server from handling events.

Hence, converting the Notes application to use asynchronous operations to access the data is going to require some surgery.

What we're going to do is as follows:

- Convert the model API to use asynchronous functions.
- Refactor the views to not have asynchronous functions.
- Convert the routers to handle all the asynchronous stuff before handing data to the view templates.

The first step is the `index.ejs` view, and the router in `routes/index.js`. Because EJS does not support asynchronous code in the template, we must do all the asynchronous data retrieval first and then render the template.

Change `views/index.ejs` to read as follows:

```
<% include top %>
<%
if (notes) {
  notes.forEach(function(note) {
    %><p><%= note.key %>:
    <a href="/noteview?key=<%= note.key %>"><%= note.title %></a>
    </p><%
  });
}
%>
<% include bottom %>
```

This template is to be given an array of the `key` and `title` values, with which it makes a list of links.

Change `routes/index.js` to read as follows:

```
var notes = undefined;
exports.configure = function(params) {
  notes = params.model;
}
exports.index = function(req, res) {
  notes.titles(function(err, titles) {
    if (err) {
      res.render('showerror', {
        title: "Could not retrieve note keys from data store",
        error: err
      });
    } else {
      res.render('index', { title: 'Notes', notes: titles });
    }
  });
};
```

The `configure` function is used to inject the model into the router. We'll go over this shortly.

It is the `notes.titles` function, which we'll see soon, that constructs the array of values required by the `index.ejs` template.

For error conditions, the `showerror.ejs` template is rendered instead. In this case we have a simple error page, but for a polished application you'll want to implement a nicer error page. Something like birds lifting a whale out of the ocean, maybe?

Create a file named `views/showerror.ejs` containing the following lines of code:

```
<% include top %>
<%= error %>
<% include bottom %>
```

This simply renders the error object on the page.

Injecting the model configuration into routers

The `configure` function shown in the previous section is used to inject the model being used into router modules. The purpose is to separate model code from the router modules.

Express itself doesn't provide a set best practice for the location of model code, connecting the model to router code, or configuring database connections in the models. We can observe, though, that the `app.js` module already has the purpose of configuring the application. The choice of data model, and database configuration, is an aspect of application configuration. Therefore it is `app.js` which should take care of wiring together the model, database configuration, and routers.

Each router module will export a `configure` function. It receives the module to use for the data model, and in the future may be extended to support other configuration parameters. The router modules are to use this model for retrieving data.

The data model module gets passed in via `params.model`. Note that the object we get from `require('moduleName')` is no different from any other JavaScript object. That means it can be passed around and manipulated just like other objects, like we're doing here. The `require` statement, which instantiates the module, will be performed in `app.js`, and then passed into each router module.

Later on we'll see the code in `app.js` which connects data models, database configurations, and the router modules.

The notes router

The notes router (`routes/notes.js`) is getting a lot of changes. We'll go over those changes function by function:

```
var notes = undefined;
exports.configure = function(params) {
  notes = params.model;
}
```

This is the same `configure` function we just discussed:

```
var readNote = function(key, res, done) {
  notes.read(key,
    function(err, data) {
      if (err) {
        res.render('showerror', {
          title: "Could not read note " + key,
          error: err
        });
        done(err);
      } else done(null, data);
    });
}
```

This function implements code that would otherwise be repeated several times throughout the module. Namely, reading a note out of the data store. This function in turn makes the other functions more compact.

```
exports.view = function(req, res, next) {
  if (req.query.key) {
    readNote(req.query.key, res, function(err, data) {
      if (!err) {
        res.render('noteview', {
          title: data.title,
          notekey: req.query.key,
          note: data
        });
      }
    });
  } else {
    res.render('showerror', {
      title: "No key given for Note",
      error: "Must provide a Key to view a Note"
    });
  }
}
```

This function handles viewing a single note. It retrieves the note, and renders it. The view used to render the note is unchanged from the one used previously:

```
exports.save = function(req, res, next) {
  ((req.body.dcreate === "create")
    ? notes.create : notes.update
  )(req.body.notekey, req.body.title, req.body.body,
    function(err) {
      if (err) {
        // show error page
        res.render('showerror', {
          title: "Could not update file",
          error: err
        });
      } else {
        res.redirect('/noteview?key='+req.body.notekey);
      }
    });
}
```

The `save` function is a special case because it can be used either to create a new note or to update an existing one. We use the `dcreate` parameter in the form. Depending on that value, the code either calls `notes.create` or `notes.update`. This is a necessary distinction because for some models, like SQL databases, the `create` and `update` functions are very different.

```
exports.add = function(req, res, next) {
  res.render('noteedit', {
    title: "Add a Note",
    dcreate: true,
    notekey: "",
    note: undefined
  });
}
```

The `add` function is unchanged from the one in the previous chapter. This renders the page with which the user creates a new note:

```
exports.edit = function(req, res, next) {
  if (req.query.key) {
    readNote(req.query.key, res, function(err, data) {
      if (!err) {
        res.render('noteedit', {
          title: data ?
            ("Edit " + data.title) : "Add a Note",
          dcreate: false,
          notekey: req.query.key,
          note: data
        });
      }
    });
  } else {
    res.render('showerror', {
      title: "No key given for Note",
      error: "Must provide a Key to view a Note"
    });
  }
}
```

The `edit` function sets up an editor window to either create a note or edit an existing one:

```
exports.destroy = function(req, res, next) {
  if (req.query.key) {
    readNote(req.query.key, res, function(err, data) {
      if (!err) {
```

```

        res.render('noteddelete', {
            title: data.title,
            notekey: req.query.key,
            note: data
        });
    }
    });
} else {
    res.render('showerror', {
        title: "No key given for Note",
        error: "Must provide a Key to view a Note"
    });
}
}

```

The `destroy` function sets up a window that asks the user whether they really want to delete this note:

```

exports.dodestroy = function(req, res, next) {
    notes.destroy(req.body.notekey, function(err) {
        if (err) {
            res.render('showerror', {
                title: "Could not delete Note "
                    + req.body.notekey,
                error: err
            });
        } else {
            res.redirect('/');
        }
    });
}

```

If the user confirms they want to delete the note, the `dodestroy` function gets called. This is where we actually call `notes.destroy` to delete the note.

Storing notes in files

An often underlooked database engine is the filesystem. While filesystems don't have the sort of query features supported by database engines, they are a reliable place to store files. The notes schema is simple enough that the filesystem can easily serve as its data storage layer.

Let's start by creating a directory for the model code:

```
$ mkdir models-fs
$ touch models-fs/notes.js
```

Then in `models-fs/notes.js` we start adding code:

```
var fs    = require('fs');
var path  = require('path');
var async = require('async');
var _dirname = undefined;
exports.connect = function(dirname, callback) {
  _dirname = dirname;
  callback();
}
exports.disconnect = function(callback) {
  callback();
}
```

The `connect` and `disconnect` functions will be used in each model to connect to or disconnect from the database. The first parameter to the `connect` function is meant to be data required to connect with, say, a database engine, such as the connection URL. In this case it is simply the directory name where notes are stored.

We'll be storing the notes in a directory, one note per file, with the file containing JSON formatted data:

```
exports.create = function(key, title, body, callback) {
  fs.writeFile(path.join(_dirname, key + '.json'),
    JSON.stringify({
      title: title, body: body
    }), 'utf8',
    function(err) {
      if (err) callback(err);
      else callback();
    });
}
exports.update = exports.create;
```

As for the in-memory module, the `create` and `update` functions are the same. It simply writes the JSON formatted data to a file. By adding the `.json` extension, the system can present it more correctly, for example, with some syntax coloring in text editors.

The `path.join` function takes care of forming the pathname correctly even on Windows, where the slashes go the other way around. It comes from the core `path` module which provides quite a few useful functions:

```
exports.read = function(key, callback) {
  fs.readFile(path.join(_dirname, key + '.json'), 'utf8',
    function(err, data) {
      if (err) callback(err);
      else {
        callback(undefined, JSON.parse(data));
      }
    });
}
```

Likewise, we simply read the data from the file and convert it into an object:

```
exports.destroy = function(key, callback) {
  fs.unlink(path.join(_dirname, key + '.json'),
    function(err) {
      if (err) callback(err);
      else callback();
    });
}
```

To destroy the note we simply delete it from the filesystem:

```
exports.titles = function(callback) {
  fs.readdir(_dirname, function(err, filez) {
    if (err) callback(err);
    else {
      var thenotes = [];
      async.eachSeries(filez,
        function(fname, cb) {
          var key = path.basename(fname, '.json');
          exports.read(key, function(err, note) {
            if (err) cb(err);
            else {
              thenotes.push({
                key: fname, title: note.title });
              cb();
            }
          });
        },
      );
    }
  });
}
```



```
        function(err) {
            if (err) callback(err);
            else callback(null, thenotes);
        });
    }
}
});
}
```

The `titles` function is written for the needs of the Notes home page, and produces a list of objects containing the `key` and `title` parameter for every note. Because we don't have a typical database, we can't do a query to retrieve those columns from a table. Instead we read in each note, using the `read` function, and then construct the array.

Configuring `app.js`

Remember that our strategy is to configure the application in `app.js`, wiring up the data model, database configuration, and routers. Add this code into `app.js` just before setting up the routes:

```
var model = require('./models-fs/notes');
model.connect("./Notes", function(err) {
    if (err) throw err;
});
[ routes, notes ].forEach(function(router) {
    router.configure({ model: model });
});
```

What we're doing is as follows:

- Pulling in the data model module
- Connecting the data model with its data storage
- Connecting the configured data model to all routers

In this case the model being used is the filesystem model we just discussed. The note files will be stored in a directory named `Notes`.

Because the model code used the `async` module, we need to declare it as a dependency in `package.json` as so:

```
"dependencies": {
  "express": "3.1.0",
  "ejs": "*",
  "async": "*"
}
```

Then ensure the `async` module gets installed using the following command:

```
$ npm install
```

Now, create the `Notes` directory:

```
$ mkdir Notes
```

And then run the server:

```
$ node app.js
```

```
Express server listening on port 3000
```

It will behave exactly the same as the prior version of the `Notes` application, but with three differences. First, you can kill the server and restart it and see exactly the same notes. Second, you can edit the notes with a regular text editor like `vi`. Third, you can run multiple servers and still see exactly the same notes (refer to the previous chapter to see how to do so).

Storing notes with the LevelUP data store

To get started with actual databases, let's look at an extremely lightweight, small footprint database engine, **LevelUP**. This is a Node-friendly wrapper around the LevelDB engine developed by Google, that's normally used in web browsers for local data persistence. It is a non-indexed, NoSQL, data store originally intended for use in browsers. The Node module, `LevelUP`, uses the LevelDB API and supports multiple backends, including `LevelDOWN` which integrates the C++ LevelDB database into Node.

It does not support simultaneous access from multiple processes. Therefore, `LevelUP` is not useful for sharing data between multiple Node instances.

To learn more about `LevelUP` see <https://github.com/rvagg/node-levelup>.

To learn more about LevelDB see <http://code.google.com/p/leveldb/>.

Installing LevelUP

Installation is this simple:

```
$ npm install levelup leveldown
```

Because `LevelDOWN` is a native code module, `npm` has to compile the module during its installation, and your computer must support C/C++ compilation. Compiling native code Node modules are done using `node-gyp`. The dependencies for that package are Python 2.7 (not Python 3.x), and a C/C++ compiler toolchain.

On Unix and Linux systems (including Mac OS X) the dependencies for node-gyp are simple to install, and may already be present.

On Windows systems, the dependencies are easy to come by and the Express version of Visual Studio works.

See <https://github.com/TooTallNate/node-gyp#installation> for more details. That page has links to the Microsoft free compiler downloads required.

LevelUP model code for Notes

Create a directory named `models-levelup`, and in that directory create a file, `notes.js`, containing the following:

```
var levelup = require('levelup');
var db = undefined;
exports.connect = function(location, callback) {
  db = levelup(location, {
    createIfMissing: true,
    valueEncoding: "json"
  }, callback);
}
exports.disconnect = function(callback) {
  db.close(callback);
}
```

This is the administrative code to set up the connection to a given database. The `location` parameter is a pathname to a directory which contains the database. By specifying `createIfMissing`, we ensure the database gets created even if we forget to do so.

LevelUP simply works with files in the filesystem, meaning it is easy to set up and maintain:

```
exports.create = function(key, title, body, callback) {
  db.put(key, { title: title, body: body }, callback);
}
exports.update = exports.create;
exports.read = function(key, callback) {
  db.get(key, callback);
}
exports.destroy = function(key, callback) {
  db.del(key, callback);
}
```

```

exports.titles = function(callback) {
  var thenotes = [];
  db.createReadStream()
    .on('data', function(data) {
      thenotes.push({
        key: data.key,
        title: data.value.title });
    })
    .on('error', function(err) {
      callback(err);
    })
    .on('end', function() {
      callback(undefined, thenotes);
    });
}

```

Most of these functions are a simple mapping to the LevelUP API. The `titles` function, however, tells us an interesting story about LevelUP. Namely, that LevelUP does not support any kind of indexing, or even a query function. What we do in the `titles` function is to read through all the items in the database, and, as we do so, assemble our array of item keys and titles.

The `createReadStream` function reads through every item in the database, offering your code a chance to operate on every item. A query function can easily be implemented on top of `createReadStream`. It would look similar to the `titles` function but takes a parameter describing what to look for.

Configuring `app.js` for LevelUP

Wiring up this model in `app.js` is this simple:

```

// LevelUp model
var model = require('./models-levelup/notes');
model.connect('./chap06', function(err) {
  if (err) throw err;
});
// Configure the routers with model
[ routes, notes ].forEach(function(router) {
  router.configure({ model: model });
});

```

Once you have this configured you can run Notes as before:

```
$ node app.js
```

Once running, you can use the application as before. You'll be able to kill the server and restart it to see that it persists the data.

Because LevelUP does not support access to the database from multiple instances, do not try the instructions for running multiple simultaneous Notes application instances.

Storing notes in SQL – SQLite3

To get started with actual databases let's look at using SQL from Node. First we'll use SQLite3, a lightweight, simple-to-set-up, database engine eminently suitable for many applications. To learn about that database engine see <http://www.sqlite.org/>.

The primary advantage of SQLite3 is it doesn't require a server, but is a self-contained, no-setup-required SQL database.

Setting up a schema with SQLite3

The first step is to make sure our database is configured. We're using this SQL table definition for the schema:

```
CREATE TABLE IF NOT EXISTS notes (
  notekey VARCHAR(255),
  title   VARCHAR(255),
  author  VARCHAR(255),
  body    TEXT
);
```

To set this up we need to run the `sqlite3` command. To do so we need to first install the `sqlite3` package. For systems with package managers (Linux, MacPorts, and so on) it is probably available as an installable package. Otherwise, source or precompiled binaries are available through the website link given earlier.

Once you have the `sqlite3` command available, you can run this command:

```
$ sqlite3 chap06.sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> CREATE TABLE IF NOT EXISTS notes (
...>     notekey VARCHAR(255),
...>     title   VARCHAR(255),
...>     author  VARCHAR(255),
...>     body    TEXT
```

```

...> );
sqlite> .schema notes
CREATE TABLE notes (
  notekey VARCHAR(255),
  title   VARCHAR(255),
  author  VARCHAR(255),
  body    TEXT
);
sqlite> ^D

```

This creates our database in the file `chap06.sqlite3`. The `.schema` command lets you confirm the database schema was set up as expected. On Windows, press `CTRL + C` to exit the `sqlite3` shell, rather than `CTRL + D`.

Model code

Create a directory for the model code:

```
$ mkdir models-sqlite3
```

And in that directory create a file named `notes.js`:

```

var sqlite3 = require('sqlite3');
sqlite3.verbose();
var db = undefined;
exports.connect = function(dbname, callback) {
  db = new sqlite3.Database(dbname,
    sqlite3.OPEN_READWRITE | sqlite3.OPEN_CREATE,
    function(err) {
      if (err) callback(err);
      else    callback();
    }
  );
}
exports.disconnect = function(callback) {
  callback();
}

```

This is the same connect and disconnect functionality we've already seen. The `sqlite3` module has many options for opening databases, including read-only:

```

exports.create = function(key, title, body, callback) {
  db.run("INSERT INTO notes ( notekey, title, body) "+
    "VALUES ( ?, ? , ? );",

```

```
        [ key, title, body ],
        function(err) {
            if (err) callback(err);
            else      callback();
        });
    }
    exports.update = function(key, title, body, callback) {
        db.run("UPDATE notes "+
            "SET title = ?, body = ? "+
            "WHERE notekey = ?",
            [ title, body, key ],
            function(err) {
                if (err) callback(err);
                else      callback();
            });
    }
}
```

Because we have a SQL database, the create and update functions have different implementations.

The `sqlite3` module uses a parameter substitution paradigm that's common in SQL programming interfaces. The programmer puts the SQL query into a string, and then places a question mark in each place it's desired to insert a value into the query string. Each question mark in the query string has to match with a value in the array provided by the programmer. The module takes care of encoding the values correctly so that the query string is properly formatted, while preventing SQL injection attacks.

The `db.run` function simply runs the SQL query it is given, and does not retrieve any data.

```
exports.read = function(key, callback) {
    db.get("SELECT * FROM notes WHERE notekey = ?",
        [ key ],
        function(err, row) {
            if (err) callback(err);
            else      callback(null, row);
        });
}
```

To retrieve data using the `sqlite3` module, you use either the `db.get`, `db.all`, or `db.each` functions. The `db.get` function used here returns the first row of the result set. The `db.all` function returns all rows of the result set at once, which can be a problem for available memory if the result set is large. The `db.each` function retrieves one row at a time, while still allowing processing of the entire result set.

In this case we just want the one row matching the note that is being requested:

```
exports.destroy = function(key, callback) {
  db.run("DELETE FROM notes WHERE notekey = ?",
    [ key ],
    function(err) {
      if (err) callback(err);
      else      callback();
    });
}
```

Deleting entries is as simple as the SQL DELETE command:

```
exports.titles = function(callback) {
  var titles = [];
  db.each("SELECT notekey, title FROM notes",
    function(err, row) {
      if (err) callback(err);
      else titles.push({
        key: row.notekey, title: row.title });
    },
    function(err, num) {
      if (err) callback(err);
      else callback(null, titles);
    });
}
```

The `db.each` function conveniently iterates over each row of the result set, calling the first function on each. This avoids the memory footprint of loading the whole result set into memory at once, and instead you process it one item at a time. Once it is done processing the rows, it calls the second function.

In this case we're collecting the data into the array in the first function, and in the second function we send that array to the caller.

Configuring app.js

The `app.js` module will require a slightly different data model configuration:

```
var model = require('./models-sqlite3/notes');
model.connect("./chap06.sqlite3", function(err) {
  if (err) throw err;
});
[ routes, notes ].forEach(function(router) {
  router.configure({ model: model });
});
```


For SQLite3, the connection information is the filename of the database.

Add the dependency in `package.json`:

```
"sqlite3": "*"

```

And ensure the `sqlite3` module is installed:

```
$ npm install

```

Note that this step involves compiling a native code module. On Windows, the compilation dependencies can be tricky, and beyond satisfying the `node-gyp` dependencies discussed earlier.

It may help to type the following command instead:

```
$ npm install sqlite3 --arch=ia32

```

It may also help to install Visual Studio 2012 in addition.

And then run the server:

```
$ node app.js

```

Express server listening on port 3000

The Notes application will behave the same as with `models-fs`, but the data will be stored in the database. You can verify the database content directly:

```
$ sqlite3 chap06.sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select * from notes;
test1|Hello, World!|Data stored in sqlite3
test2|Hi Mom|hi there
sqlite>

```

Storing notes the ORM way with the Sequelize module

There are other popular SQL databases and for each there is a module of the sort we just saw with SQLite3. These modules put the programmer close to the SQL, which can be good in the same way that driving a stick shift car is fun. Other modules provide a higher level interface and even offer the ability to use the data model with several databases.

ORM stands for **Object-Relation Mapping**, and is a methodology to map high level objects onto database tables.

The **Sequelize** module (<http://www.sequelizejs.com/>) can connect with SQLite3, MySQL, and PostgreSQL. It provides an ORM approach to these three SQL databases.

A prerequisite is to have access to a suitable database server. Most web hosting providers offer MySQL or PostgreSQL as part of the service.

Schema setup and model code

In Sequelize the schema is defined in JavaScript code you write in your module. This means schema setup is not a separate step, written in SQL, like it was with SQLite3.

To start, make a directory to hold the model:

```
$ mkdir models-sequelize
```

In that directory, create a file named `notes.js` and start entering this code:

```
var Sequelize = require("sequelize");
var Note = undefined;
module.exports.connect = function(params, callback) {
  var sequlz = new Sequelize(
    params.dbname, params.username, params.password,
    params.params);
  Note = sequlz.define('Note', {
    notekey: { type: Sequelize.STRING,
              primaryKey: true, unique: true },
    title: Sequelize.STRING,
    body: Sequelize.TEXT
  });
  Note.sync().success(function() {
    callback();
  }).error(function(err) {
    callback(err);
  });
}
```

```
    });  
  }  
  exports.disconnect = function(callback) {  
    callback();  
  }  
}
```

The connect function in this case takes an object listing parameters, rather than a connection string. That's because connecting through Sequelize requires passing an object containing parameters describing the database connection. The first three control the database name and authentication, while the fourth is an object describing the database connection to use. That object can be configured to connect with SQLite3, MySQL, and PostgreSQL databases.

With Sequelize we do not define the schema in terms of tables and columns, but in terms of an object definition. What Sequelize does is map the object attributes into columns in tables. Here we've defined a simple object with the same attributes in the other notes model implementations. Finally, the `Notes.sync()` function ensures the table is set up.

```
exports.create = function(key, title, body, callback) {  
  Note.create({  
    notekey: key,  
    title: title,  
    body: body  
  }).success(function(note) {  
    callback();  
  }).error(function(err) {  
    callback(err);  
  });  
}  
exports.update = function(key, title, body, callback) {  
  Note.find({ where: { notekey: key } }).success(function(note) {  
    if (!note) {  
      callback(new Error("No note found for key " + key));  
    } else {  
      note.updateAttributes({  
        title: title,  
        body: body  
      }).success(function() {  
        callback();  
      }).error(function(err) {  
        callback(err);  
      });  
    }  
  }).error(function(err) {  
    callback(err);  
  });  
}
```

```

        callback(err);
    });
}

```

Creating a new instance in Sequelize can be done in several ways, the simplest of which is to call an object's `create` function. That function collapses together two other functions, `build`, to create the object, and `save`, to write it to the database.

Updating an instance is a little different. First we must retrieve its entry from the database using the `find` operation. Finding an instance is done with the `find` operation, which takes an object that specifies the query to perform to locate items. The `find` operation retrieves one instance, whereas the `findAll` operation retrieves all matching instances.

There are instances where the `success` function is called, but the `note` object is empty. We account for that case by sending an error saying no note was found.

Once the instance is found, we can update its values simply with the `updateAttributes` function:

```

exports.read = function(key, callback) {
    Note.find({ where:{ notekey: key } }).success(function(note) {
        if (!note) {
            callback("Nothing found for " + key);
        } else {
            callback(null, {
                notekey: note.notekey,
                title: note.title,
                body: note.body
            });
        }
    });
}

```

Here we find the note instance, and send its data to the caller:

```

exports.destroy = function(key, callback) {
    Note.find({ where:{ notekey: key } }).success(function(note) {
        note.destroy().success(function() {
            callback();
        }).error(function(err) {
            callback(err);
        });
    });
}

```

To delete information from the database use the `destroy` function:

```
exports.titles = function(callback) {
  Note.findAll().success(function(notes) {
    var thenotes = [];
    notes.forEach(function(note) {
      thenotes.push({
        key: note.notekey, title: note.title });
    });
    callback(null, thenotes);
  });
}
```

Finally, we can use `findAll` to retrieve the data to be returned by the `titles` function, as shown here. This function, as the name implies, finds all matching objects. As used here it retrieves all objects, but it can be given a finder object to filter the list of retrieved objects.

Configuring app.js

As before, configuring the Notes application to use Sequelize is done in `app.js`, where we initialize the model code and pass the model module to the router modules.

```
var model = require('./models-sequelize/notes');
model.connect({
  dbname: "database name",
  username: "user name",
  password: "password",
  params: {
    host: 'database host name',
    dialect: 'mysql'
  }
},
function(err) {
  if (err) throw err;
});
[ routes, notes ].forEach(function(router) {
  router.configure({ model: model });
});
```

To configure Sequelize, we give an object to the `connect` method needed for initializing Sequelize, as is documented on their website. What we show here is set up for MySQL, and using the same code with SQLite3 or PostgreSQL is simply a matter of changing the parameters. Of course, to use MySQL or PostgreSQL, a prerequisite step is to set up and configure the database server, ensuring the configuration parameters to suite.

Add the dependency in `package.json`:

```
"sequelize": "*" 
```

And ensure the Sequelize module is installed:

```
$ npm install
```

And then run the server:

```
$ node app.js
```

```
Express server listening on port 3000
```

As with using the other data models, the Notes application will behave the same. You can verify what's shown on the screen by logging into the MySQL database and entering a command like this:

```
mysql> select * from Notes;
```

In addition to the columns we specified in the schema, Sequelize adds at least three more columns. The other columns (`id`, `createdAt`, and `updatedAt`) are used for Sequelize's bookkeeping.

Storing notes in MongoDB with Mongoose

MongoDB is one of the leading NoSQL databases. They describe it as a *scalable, high performance, open source, document-oriented database*. It uses JSON-style documents with no predefined rigid schema, and a large number of advanced features. You can see their website for more information and documentation: <http://www.mongodb.org>.

A prerequisite is to have a MongoDB instance available. If you do not have access to one, **mongohosting** (<http://www.mongohosting.com>), **mongohq** (<http://www.mongohq.com>), and **mongolab** (<http://www.mongolab.com>) all offer a free pricing tier to use on a trial basis.

Mongoose is one of several modules for accessing MongoDB from Node. It is an object modeling tool, meaning that your program defines schema objects describing its data, and Mongoose takes care of storage in MongoDB. It's a very powerful object modeling tool for Node and MongoDB, with embedded documents, a flexible typing system for fields, field validation, virtual fields, and more. See <http://mongoosejs.com> for more information.

The sponsor of the MongoDB project, 10gen, also sponsors the Mongoose framework.

Implementing the Notes model in Mongoose

To get started, create a directory to hold the Mongoose version of the Notes model:

```
$ mkdir models-mongoose
```

Then, create a file named `notes.js` in that directory:

```
var util      = require('util');
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var dburl = undefined;
exports.connect = function(thedburl, callback) {
  dburl = thedburl;
  mongoose.connect(dburl);
}
exports.disconnect = function(callback) {
  mongoose.disconnect(callback);
}
var NoteSchema = new Schema({
  notekey: String,
  title  : String,
  body   : String
});
mongoose.model('Note', NoteSchema);
var Note = mongoose.model('Note');
```

This handles the administrative tasks of connecting to or disconnecting from the database, and defining the schema. Like Sequelize, in Mongoose the schema is defined by creating an object to describe the schema. You provide that object to Mongoose, and it handles translating of the object data back and forth with the database.

With Mongoose, the connection is described by a URL to the MongoDB server.

The schema here is the same as we've been using in the other models for Notes:

```
exports.create = function(key, title, body, callback) {
  var newNote    = new Note();
  newNote.notekey = key;
  newNote.title   = title;
  newNote.body    = body;
  newNote.save(function(err) {
    if (err) callback(err);
    else      callback();
  });
}
```

Creating a new object instance to save it in the database is straightforward. We create an instance of the object returned from Mongoose, assign fields to it, then call `save`:

```
exports.update = function(key, title, body, callback) {
  exports.read(key, function(err, doc) {
    if (err) callback(err);
    else {
      doc.title   = title;
      doc.body    = body;
      doc.save(function(err) {
        if (err) callback(err);
        else      callback();
      });
    }
  });
}
```

Updating an existing object is a little different. First we retrieve it from the database using our `read` function, which we'll look at in a minute. Once the document object is retrieved, we assign new data to it, then call `save`:

```
exports.read = function(key, callback) {
  Note.findOne({ notekey: key }, function(err, doc) {
    if (err) callback(err);
    else      callback(null, doc);
  });
}
```


Here is our `read` function. The `findOne` function retrieves the first matching document, with the match dictated by the object passed in. If you want to retrieve all matching documents, use the `find` function. There are several other `find` functions that combine a query with either an update or delete operation:

```
exports.destroy = function(key, callback) {
  exports.read(key, function(err, doc) {
    if (err) callback(err);
    else {
      doc.remove();
      callback();
    }
  });
}
```

As for the update function, to delete a document we first have to retrieve the document, at which point we can call the object's `remove` function. This is because these operations are methods on the document object.

```
exports.titles = function(callback) {
  Note.find().exec(function(err, docs) {
    if (err) callback(err);
    else {
      if (docs) {
        var thenotes = [];
        docs.forEach(function(note) {
          thenotes.push({
            key: note.notekey, title: note.title });
        });
        callback(null, thenotes);
      } else {
        callback();
      }
    }
  });
}
```

This version of the `titles` function is similar to the one for Sequelize. We use the Mongoose `find` operation to retrieve all documents, then iterate over each creating our array containing the `key` and `title` data.

Configuring app.js

As before, configuring the Notes application to use Mongoose is done in `app.js`, where we initialize the model code and pass the model module to the router modules:

```
var model = require('./models-mongoose/notes');
model.connect("mongodb://localhost/chap06");
[ routes, notes ].forEach(function(router) {
  router.configure({ model: model });
});
```

For Mongoose, we give a string containing the URL for the MongoDB instance. Make sure to use the correct hostname for the server you're using. As before, we provide that model to the router modules.

Add the dependency in `package.json`:

```
"mongoose": "*"

```

And ensure the Mongoose module is installed:

```
$ npm install

```

And then run the server:

```
$ node app.js

```

```
Express server listening on port 3000

```

As with using `model-sequelize`, the behavior of the Notes application will be the same. You can verify the database content by logging into the database using the `mongo` client and entering a command like this:

```
> use chap06
switched to db chap06
> db.notes.find();
{ "body" : "Hello World!", "title" : "hi there", "notekey" : "hithere",
  "_id" : ObjectId("5137d8f08691790000000001"), "__v" : 0 }
{ "__v" : 0, "_id" : ObjectId("5137da09672cb90000000001"), "body" :
  "hihihi 2 3 4", "notekey" : "gogoooo", "title" : "goooooooood" }
>
```

Summary

We've come a long way in this chapter, rewriting the Notes application to use several database engines. However, we've only touched the surface of options for accessing databases and data storage engines from Node. The Node wiki contains a list of database engines and a large number of modules that you can find at <https://github.com/joyent/node/wiki/modules#wiki-database>, covering a whole gamut of engines and models for using databases.

By abstracting the model implementations correctly, we were able to easily switch data storage engines while not changing the rest of the application.

By focusing the model code on the purpose of storing data, both the models and the application should be easier to test. Namely, the application could be tested with a mock data module that provides known predictable notes that can be checked predictably. We'll look at this in more depth in *Chapter 9, Unit Testing*.

In the next chapter we'll focus on deploying our application for real use by real people.

7

Multiuser Authorization, Deployment, Scaling, and Hosting

Now that our Notes application can save its data in a database, we can think about the next phase of making this a real application. The three aspects of this that we'll discuss in this chapter are:

- User authentication, to control who can edit notes
- Deploying a Node application to a production server
- Scaling the application to match traffic

User authentication

It seems so natural to *log in* to a website to use its services. We do it everyday, and even trust our banking and investments organizations with the information accessed, by logging in to websites. But HTTP is a stateless protocol, and a web server or a web application cannot tell much about one HTTP request versus another. Because HTTP is stateless, that means HTTP requests do not natively carry **state**, such as whether the user driving the web browser is logged in, their identity, or even whether the HTTP request was initiated by a human being.

The typical method for user authentication is to send a cookie, containing a token that carries user identity, to the browser. The cookie needs to contain data identifying the browser, and whether that browser is logged in. The cookie will then be sent with every request, letting the application track which user account is associated with the browser.

With Express and Connect, the best way to do this is with the `session` middleware. It stores data as a cookie, and retrieves that data when the browser sends requests. It is easy to configure, but is not a complete solution for user authentication. There are several add-on modules that handle user authentication, and some even support authenticating users against third-party websites, such as Facebook or Twitter.

Two packages appear to be leading the pack in user authentication, Passport (<http://passportjs.org/>) and Everyauth (<http://everyauth.com/>). Both support a long list of services against which to authenticate, making it easy to develop a website that lets users sign up with credentials from another website.

We will use the passport system to authenticate users against our own list of user accounts. Do not take user authentication and storing data about your users lightly. Do you want your website to be blasted across the Internet as the latest source of stolen identity data?

Let's get started!

Changes in app.js

To implement user authentication in the Notes application we'll use the **passport-local** strategy to create our own list of user accounts.

Passport uses, what it calls, **Strategies**, to authenticate users against the various services. The **Local** strategy is one where the authentication data is stored locally to the application, rather than consult another service. A long list of add-on modules for Passport implement authentication against other services, such as Twitter or GitHub.

Start by duplicating the code used in the previous chapter. We'll be modifying a few of the files.

At the top of `app.js`, add these require calls:

```
var flash = require('connect-flash');
var users = require('./routes/users');
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;
```

Next, configure some functions in passport to serialize data in and out of the session. These functions will be implemented in the user routing module, which we'll go over in a little bit.

```
passport.serializeUser(users.serialize);
passport.deserializeUser(users.deserialize);
passport.use(users.strategy);
```

Next we have a number of things to add in `app.js`:

```
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, '/views'));
app.set('view engine', 'ejs');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.cookieParser());
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(express.session({ secret: 'keyboard cat' }));
// Initialize Passport! Also use passport.session() middleware,
// to support persistent login sessions (recommended).
app.use(flash());
app.use(passport.initialize());
app.use(passport.session());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));
```

Because the `session` middleware stores its data as a cookie, we need the `cookieParser` middleware. That middleware is added before the `session` middleware, to ensure cookie data is parsed before the `session` middleware processes the request. Finally the `Passport` middleware is added after the `session` middleware, because `Passport` relies on `session` functionality.

Because we referenced some new modules above, add the following package dependencies in `package.json`:

```
'connect-flash': '*',
'passport': '*',
'passport-local': '*'
```

Then make sure these modules are installed by running `npm install` again.

Next we need to set up the data models. It will also be useful to change how we manage the database connection parameters object.

We're going to be reusing the same database connection parameters in multiple places. Not only in `app.js`, but in other scripts. In order to avoid repeating these parameters multiple times, we'll use a configuration file to store the parameters in one place and reference them from multiple places.

The principle is *don't repeat yourself* and by not repeating the same code in multiple places, the application is more maintainable.

Node doesn't have a defined configuration file format to use. While there are several available modules supporting configuration data files, we're going to do something different. Recall that the `module.exports` object in a Node module is returned from the `require` statement, and that this object can contain anything.

Create a file named `sequelizeize-params.js` in the same directory as `app.js`, containing the following:

```
module.exports = {
  dbname: '..', username: '..', password: '..',
  params: {
    host: 'mysql.example.com',
    dialect: 'mysql'
  }
};
```

This is a simple JavaScript object that will be returned whenever we require the `sequelizeize-params` module.

Then in `app.js`, you can add this code prior to initializing the data models:

```
var seqConnectParams = require('./sequelizeize-params');
```

Then, when initializing the data modules simply refer to this object, as follows:

```
var notesModel = require('./models-sequelize/notes');
notesModel.connect(seqConnectParams,
  function(err) {
    if (err) throw err;
  });
```

The Notes model remains the same as before; we're simply changing how it's initialized.

We also want to add a model to store the user authentication data. We will initialize that model like so, in `app.js`:

```
var usersModel = require('./models-sequelize/users');
usersModel.connect(seqConnectionParams,
  function(err) {
    if (err) throw err;
  });
users.configure({
  users: usersModel,
  passport: passport
});
```

The `users` module, which we'll look at in a minute, specifies a schema for a table which we'll store in the same database as the `Notes` table. We could store the users in a separate database from the `Notes`. By having them in the same database it's possible to make a `JOIN` query, if necessary.

Finally we need to add some routes into `app.js` and reconfigure existing ones:

```
app.get('/', routes.index);
app.get('/noteview', notes.view);
app.get('/noteadd', users.ensureAuthenticated, notes.add);
app.get('/noteedit', users.ensureAuthenticated, notes.edit);
app.get('/notedestroy', users.ensureAuthenticated, notes.destroy);
app.post('/notedestroy', users.ensureAuthenticated, notes.
  ddestroy);
app.post('/notesave', users.ensureAuthenticated, notes.save);
app.get('/account', users.ensureAuthenticated, users.doAccount);
app.get('/login', users.doLogin);
app.post('/login', passport.authenticate('local', {
  failureRedirect: '/login', failureFlash: true
}), users.postLogin);
app.get('/logout', users.doLogout);
```

Two things are different in these routes from the ones used in the previous version of `Notes`. First is the presence of `users.ensureAuthenticated` and second is some new routes.

The new routes handle the login and logout steps, as well as a page to show the user account information.

The `users.ensureAuthenticated` function is what Express calls `route middleware`. The routing function can invoke a chain of functions, with all but the last of those functions serving as `route middleware`. The `route middleware` functions perform processing steps before the final function.

In this case, the `users.ensureAuthenticated` function is required to authenticate the user based on session information, and if the user is not authenticated, to redirect their browser to the `/login` page. You'll note that each route using the `users.ensureAuthenticated` function is one that rightly requires an authenticated user.

The purpose of doing this is to protect certain parts of the application by requiring that the user is logged in. A more extensive application might have multiple levels of access to check, but for the `Notes` application, all we want to know is whether they're logged in.

The Sequelize-based users model

In the `models-sequelize` directory, create a file named `users.js`, containing the following code:

```
var util      = require('util');
var Sequelize = require('sequelize');
var sequelize = undefined;
var User      = undefined;
module.exports.connect = function(params, callback) {
  sequelize = new Sequelize(params.dbname, params.username,
                           params.password, params.params);
  User = sequelize.define('User', {
    id: {
      type: Sequelize.INTEGER,
      primaryKey: true, unique: true },
    username: { type: Sequelize.STRING, unique: true },
    password: Sequelize.STRING,
    email: Sequelize.STRING
  });
  User.sync().success(function() {
    callback();
  }).error(function(err) {
    callback(err);
  });
}
exports.disconnect = function(callback) {
  callback();
}
```

These functions set up the straightforward user schema. The user model functions are very similar to the Notes model functions we looked at in the previous chapter.

```
module.exports.findById = function(id, callback) {
  User.find({ where: { id: id } }).success(function(user) {
    if (!user) {
      callback('User ' + id + ' does not exist');
    } else {
      callback(null, {
        id: user.id, username: user.username,
        password: user.password, email: user.email
      });
    }
  });
}
```

```

module.exports.findByUsername = function(username, callback) {
  User.find({where:{username: username}}).success(function(user) {
    if (!user) {
      callback('User ' + username + ' does not exist');
    } else {
      callback(null, {
        id: user.id, username: user.username,
        password: user.password, email: user.email
      });
    }
  });
}

```

Support finding user information either by their id or username.

```

module.exports.create = function(id, username,
                                password, email, callback) {
  User.create({
    id: id,
    username: username,
    password: password,
    email: email
  }).success(function(user) {
    callback();
  }).error(function(err) {
    callback(err);
  });
}

module.exports.update = function(id, username,
                                password, email, callback) {
  User.find({ where: { id: id } }).success(function(user) {
    user.updateAttributes({
      id: id,
      username: username,
      password: password,
      email: email
    }).success(function() {
      callback();
    }).error(function(err) {
      callback(err);
    });
  });
}

```

The functions of creating a user account record or updating an existing one are similar to those for the Notes model.

Routing module for the login, logout, and account pages

In the routes directory, create a file named `users.js`, containing the following. If you still have a `user.js` module created by Express, delete that file.

```
var LocalStrategy = require('passport-local').Strategy;
var users        = undefined;
var passport     = undefined;
exports.configure = function(params) {
  users        = params.users;
  passport     = params.passport;
}
```

The `configure` function receives configuration information—not just the user's data model, but a reference to the `passport` module. The latter allows this router module to call the `passport` module to perform some authentication functions.

```
module.exports.serialize = function(user, done) {
  done(null, user.id);
}
module.exports.deserialize = function(id, done) {
  users.findById(id, function (err, user) {
    done(err, user);
  });
}
```

As we noted in the changes to `app.js`, these two functions store identity information in the session cookie. Only the user ID is stored in the session data, and as we saw with the model implementation, the `id` value is enough for retrieving user data from the user model.

```
module.exports.strategy = new LocalStrategy(
  function(username, password, done) {
    process.nextTick(function () {
      users.findByUsername(username, function(err, user) {
        if (err) { return done(err); }
        if (!user) { return done(null, false, {
          message: 'Unknown user ' + username }); }
        if (user.password !== password) {
          return done(null, false, {
            message: 'Invalid password' }); }
        return done(null, user);
      })
    });
  }
);
```

Here we initialize the authentication strategy used by the application. This strategy is referenced, as we saw earlier, in `app.js`. We've implemented our own list of user records and it is this function where we check whether a correct password was entered, or not.

```
module.exports.ensureAuthenticated = function(req, res, next) {  
  if (req.isAuthenticated()) { return next(); }  
  return res.redirect('/login');  
}
```

This is our route middleware function which gets used from `app.js` on every route requiring an authenticated user. The `req.isAuthenticated` function is provided by passport, giving us an easy way to check if the browser is logged in. If the user is not logged in, they're redirected to the login form.

```
module.exports.doAccount = function(req, res){  
  res.render('account', {  
    title: 'Account information for ' + req.user.username,  
    user: req.user  
  });  
}
```

This is the route handler for the `/account` page, and simply renders information about the user account.

```
module.exports.doLogin = function(req, res){  
  res.render('login', {  
    title: 'Login to Notes',  
    user: req.user,  
    message: req.flash('error')  
  });  
}  
module.exports.postLogin = function(req, res) {  
  res.redirect('/');  
}
```

The `doLogin` function simply renders a login form for the `/login` page. When that form is posted (POST) on `/login`, the `postLogin` function is called if the user is authenticated. This route (POST on `/login`) is configured in `app.js`, to first call Passport to do the authentication, and second to call the `postLogin` function if the authentication succeeded. Hence, all this function needs to do is redirect the user to the home page.

The `req.flash` function is used to show messages to the user. We get this function from the `connect-flash` module loaded in `app.js`. It stores messages in the session object, and then the message is cleared after it is displayed to the user.

```
module.exports.doLogout = function(req, res){  
    req.logout();  
    res.redirect('/');  
}
```

Finally, this handles the `/logout` page. It tells Passport to clear out the authentication information from the browser, which is then redirected to the home page.

To support the user's routing module, a few new views are required, as well as modifications to existing ones.

It's helpful for `top.ejs` to show a link for logging in, and to show who is logged in.

```
<html>  
<head>  
    <title><%= title %></title>  
    <link rel='stylesheet' href='/stylesheets/style.css' />  
</head>  
<body>  
    <h1><%= title %></h1>  
    <div class='navbar'>  
    <p><a href='/'>home</a>  
        | <a href='/noteadd'>ADD Note</a>  
        <% if (user) { %>  
        | <a href='/logout'>Log Out</a>  
        | logged in as <a href='/account'><%= user.username %></a>  
        <% } else { %>  
        | <a href='/login'>Log in</a>  
        <% } %>  
    </p>  
</div>
```

The changes are in the list of links that form the navigation bar. First we check if a user object is provided, and then display different links depending on whether the user is logged in or not.

In the views directory, create a file, named `account.ejs`, containing:

```
<% include top %>  
<p>Name: <%= user.username %> (id: <%= user.id %>)</p>  
<p>E-Mail: <%= user.email %></p>  
<% include bottom %>
```

This is used by the `doAccount` function to show account information for the logged-in user. Notice that `top.ejs` includes a link to the `/account` page, which ends up rendering this view.

Now create another views template, named `login.ejs`, containing:

```
<% include top %>
<form method='POST' action='/login'>
  <p>User name: <input type='text' name='username'></p>
  <p>Password: <input type='text' name='password' /></p>
  <p><input type='submit' value='Submit' /></p>
</form>
<% include bottom %>
```

This is a simple login form, and is rendered by the `doLogin` function which in turn is called for the `/login` route. When this form is submitted, it is executed by the `app.post` route for `/login`. That route first calls on Passport to check the login credentials, and if successful, ends up calling the `postLogin` function we discussed earlier.

Because of the changes to templates in the `views` directory, we have to make some changes to both, the `routes/index.js` and the `routes/notes.js` modules. Because the `views/top.ejs` template now requires a user object, every call to `res.render` must include a user object.

In `routes/index.js`, change the `index` function to this:

```
exports.index = function(req, res) {
  notes.titles(function(err, titles) {
    if (err) {
      res.render('showerror', {
        title: "Could not retrieve note keys from data store",
        user: req.user ? req.user : undefined,
        error: err
      });
    } else {
      res.render('index', {
        title: 'Notes',
        user: req.user ? req.user : undefined,
        notes: titles
      });
    }
  });
};
```

The important line is this:

```
user: req.user ? req.user : undefined,
```

This ensures that the `user` object exists in the template, and is either the one created by the Passport library or else is undefined.

Next, let's start editing the `routes/notes.js` module. We'll be adding similar lines to several places in that module.

Change the `readNote` function to this:

```
var readNote = function(key, user, res, cb) {
  notes.read(key,
    function(err, data) {
      if (err) {
        res.render('showerror', {
          title: "Could not read note " + key,
          user: user ? user : undefined,
          error: err
        });
        cb(err);
      } else cb(null, data);
    });
}
```

We've extended its parameter list to include the `user` object, and are passing that to `res.render`.

Now change the `view` function to this:

```
exports.view = function(req, res, next) {
  var user = req.user ? req.user : undefined;
  if (req.query.key) {
    readNote(req.query.key, user, res, function(err, data) {
      if (!err) {
        res.render('noteview', {
          title: data.title,
          user: user,
          notekey: req.query.key,
          note: data
        });
      }
    });
  } else {
    res.render('showerror', {
```

```
        title: "No key given for Note",
        user: user,
        error: "Must provide a Key to view a Note"
    });
}
}
```

We're setting up a local user variable, and passing that into a few places.

Now change the edit function to this:

```
exports.edit = function(req, res, next) {
    var user = req.user ? req.user : undefined;
    if (req.query.key) {
        readNote(req.query.key, user, res, function(err, data) {
            if (!err) {
                res.render('noteedit', {
                    title: data ?
                        ("Edit " + data.title) : "Add a Note",
                    user: user,
                    docreate: false,
                    notekey: req.query.key,
                    note: data
                });
            }
        });
    } else {
        res.render('showerror', {
            title: "No key given for Note",
            user: user,
            error: "Must provide a Key to view a Note"
        });
    }
}
```

This is the same sort of change as for the view function.

Then finally change the destroy function to this:

```
exports.destroy = function(req, res, next) {
    var user = req.user ? req.user : undefined;
    if (req.query.key) {
        readNote(req.query.key, user, res, function(err, data) {
            if (!err) {
                res.render('notedelete', {
                    title: data.title,
```



```
        user: user,
        notekey: req.query.key,
        note: data
    });
    }
    });
} else {
    res.render('showerror', {
        title: "No key given for Note",
        user: user,
        error: "Must provide a Key to view a Note"
    });
}
```

And again this is the same sort of change. All three of these functions follow a similar pattern.

The `req.user` variable is provided by `Passport`. By doing it this way we provide a `user` object to the views, but if nobody is logged in, its value is `undefined`. Then, the `if` statement in `top.ejs` detects this condition to display the correct links for users who are logged in, or not.

Initializing the user table

It's fairly normal for a web application to have a whole administrative backend, one of whose functions is to control user accounts and user registration. We're going to skip all that, but we still need to have some users listed in the user table. Instead we'll write a little script to add a couple of user accounts.

In the `models-sequelize` directory, create a file named `setup.js`, containing:

```
var users = require('./users');
users.connect(require('../sequelize-params'),
    function(err) {
        if (err) throw err;
        else {
            users.create('1', 'bob', 'secret',
                'bob@example.com', function(err) {
                    if (err) throw err;
                    else {
                        users.create('2', 'joe', 'birthday',
                            'joe@example.com', function(err) {
                                if (err) throw err;
                            });
                    }
                });
        }
    })
```

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

Then simply run this script at the command line:

```
$ node models-sequelize/setup.js
```

And then inspect the users table that results:

```
mysql> select * from Users;  
..  
2 rows in set (1.23 sec)
```

Running the Notes application

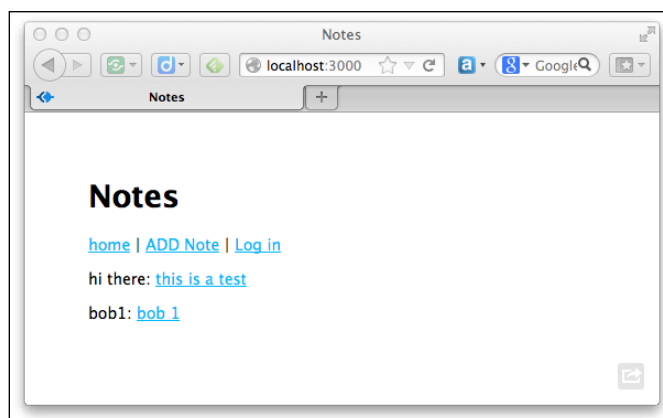
That was a lot of code, but we're now ready to run the application. What we've done is to use the `passport` package for user authentication, and add a new data model (users) to store data about our users.

We focused solely on using `Sequelize` in this chapter. Using a different database will mean implementing another users model, using that database engine, of course. It's also possible to mix up the database engines. For example, you might store the Notes in a MongoDB database, because it is a document-oriented system, and store users data in an SQL database, that gets reused by multiple applications.

Now we can run the application:

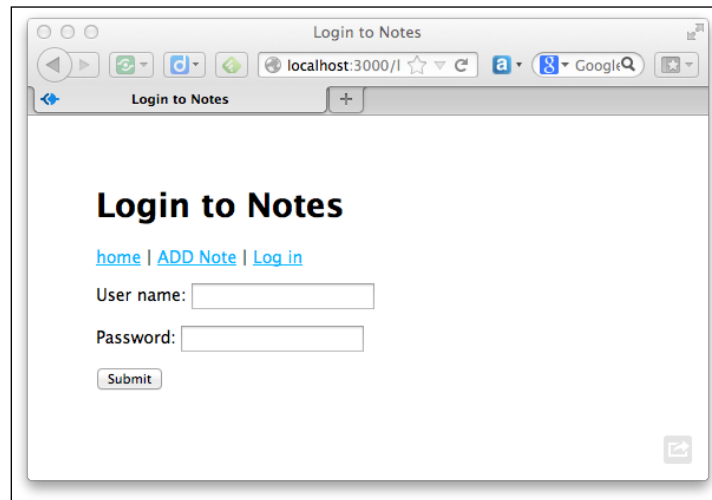
```
$ node app.js
```

Upon opening `http://localhost:3000/` in your browser, you're greeted by this screen:



You might, or might not, have notes that are left over from previous runnings of the Notes application. What's new is the **Log in** link shown, because you are not logged in. The presence of that link indicates that if you refer back to the `top.ejs` view, we are not logged in.

If you click on any of the links whose route goes through the `users.ensureAuthenticated` function, such as the **ADD Note** link, you'll be sent to the login screen. You can view each note while in the logged-out state, but you just can't take any action that requires being an authenticated user.



We already went over the authentication code, but now that it's running, we can go over it step-by-step.

We arrived on the login screen because the middleware function redirected the browser to the `/login` URL:

```
module.exports.ensureAuthenticated = function(req, res, next) {  
  if (req.isAuthenticated()) { return next(); }  
  return res.redirect('/login');  
}
```

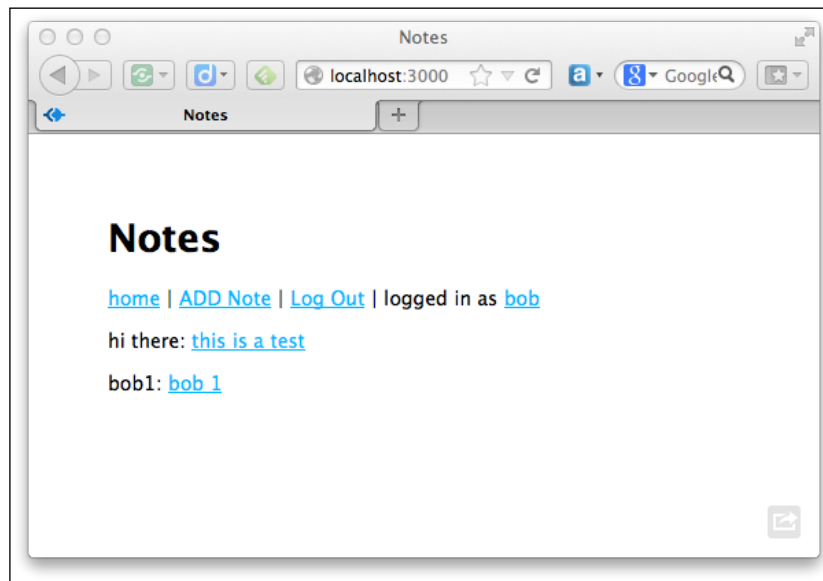
The login page is rendered from this route, to render the `login.ejs` template:

```
app.get('/login', users.doLogin);
```

When the login form is submitted, this route is executed:

```
app.post('/login', passport.authenticate('local', {
  failureRedirect: '/login', failureFlash: true
})), users.postLogin);
```

If the credentials are incorrect, you'll get an error page. If you enter correct credentials for one of the users, you will be logged in and the home screen changes to this:



The screen now shows that we are logged in, because the code in the `top.ejs` view rendered the logged-in links. The links whose routes use the `users.ensureAuthenticated` function will now work as expected.

Clicking on the **Log Out** link will return you to the logged-out state. It does so by executing this route:

```
app.get('/logout', users.doLogout);
```

Executing this function erases the authentication, and redirects to the home page.

```
module.exports.doLogout = function(req, res){
  req.logout();
  res.redirect('/');
}
```

Our Notes application can now support multiple users and prevent unauthenticated editing of the notes.

There are many ways to extend this:

- Each note could be owned by a specific user
- Users could have different permission levels; for example, an editor could edit any note, while others could only edit their own
- User authentication could rely on a third-party service rather than our own user table
- A user authenticated with a third-party service could have an option to post the note to that service, such as Twitter or Facebook

Deploying Notes on Debian

Now that we have the Notes application protected with user authentication, it's more safe to deploy it to a public server. So far we've been running the server application on the command line. This is great for debugging and development, but horrible for deployment on a real server for real users.

Before we get started, it's worth pointing out that the Notes application takes minimal precautions against attacks. It probably has many vulnerabilities, and the passwords are not securely stored.

The normal best practice is to start a background process (daemon) to manage the server process. Unfortunately, the specifics of this vary widely, from one operating system to another.

The traditional way is that the `init` daemon manages background processes using scripts in the `/etc/init.d` directory. On Fedora and Red Hat that's still the process, but other operating systems use other daemon managers, such as `upstart` or `launchd`. Generally speaking, these scripts start a background process at system boot, and let the administrator stop, start, or restart each background process individually.

The Nodes project does not include scripts for any operating system to manage a Nodes background process. It can be argued that it would be out of place for Nodes to include such scripts, and instead, Nodes server applications should include such scripts.

Web servers have to be reliable (for example, auto-restarting on crashes), manageable (integrate well with system management practices), observable (saving `STDOUT` to logfiles), and so on. Nodes is more like a construction kit with the pieces and parts for building servers, and is not a complete polished server itself. Implementing a complete web server based on Nodes means scripting to integrate with the background process management on your OS, implementing the logging features you need, implementing security practices or defenses against bad behavior such as denial of service attacks, and much more.

To demonstrate a little bit of what's involved, let's use `forever`, along with an LSB-style init script (<http://wiki.debian.org/LSBInitScripts>), to implement a background server process to manage the Notes application. The server being used is Debian 6 in a virtual machine with Node and NPM installed in `/usr/local`. Notes is installed in `/usr/local/notes`. While that location is good enough for this demonstration, you may want to use a better location under `/var` filesystem instead, if only for better compliance with LSB practices.

The `forever` tool (<https://github.com/nodejitsu/forever>) can monitor a set of processes, restarting any that crash. It also has some administrative functionality to manage the background processes it starts. It is installed this way:

```
$ sudo npm install -g forever
```

We'll be using `forever` from an init script, which we install as `/etc/init.d/node`, containing the following shell script:

```
#!/bin/sh -e
set -e
PATH=/usr/local/bin:/bin:/usr/bin:/sbin:/usr/sbin
DAEMON=/usr/local/notes/app.js
case '$1' in
    start) forever start $DAEMON ;;
    stop)  forever stop  $DAEMON ;;
    force-reload|restart)
        forever restart $DAEMON ;;
    *) echo 'Usage: /etc/init.d/node {start|stop|restart|force-
        reload}'
        exit 1
    ;;
esac
exit 0
```

Then on Debian we initialize the script this way:

```
# sudo chmod +x /etc/init.d/node
# sudo /usr/sbin/update-rc.d node defaults
update-rc.d: using dependency based boot sequencing
insserv: warning: script 'node' missing LSB tags and overrides
```

This sets things so that the `/etc/init.d/node` script is invoked on reboot and shutdown to start or stop the background process. During boot up or shutdown, each init script is executed, and its first argument is either `start` or `stop`.

We can run the `init` script by hand:

```
$ sudo /etc/init.d/node start
info: Forever processing file: /usr/local/notes/app.js
```

Because our `init` script uses `forever`, we can ask it the status of all processes it has started:

```
$ sudo forever list
info: Forever processes running
data: uid command script forever pid logfile uptime
data: [0] MGoC /usr/local/bin/node /usr/local/notes/app.js 27502
27504 /root/.forever/MGoC.log 0:0:1:22.151
```

With the server still running and managed by `forever`, we have these processes:

```
# ps ax | grep node
27502 ? Ssl 0:00 /usr/local/bin/node /usr/local/lib/node_modules/forever/
bin/monitor /usr/local/notes/app.js
27504 ? Sl 0:01 /usr/local/bin/node /usr/local/notes/app.js
27520 pts/2 S+ 0:00 grep node
```

The Notes application will run and behaves the same as before.

When you're done playing with this you can shut it down this way:

```
# /etc/init.d/node stop
info: Forever stopped process:
data: uid command script forever pid logfile uptime
[0] MGoC /usr/local/bin/node /usr/local/notes/app.js 27502 27504 /
root/.forever/MGoC.log 0:0:7:41.365
# forever list
info: No forever processes running
```

Scaling to use all cores on multi-core servers

The next phase of deploying an application for production use is ensuring that you're fully utilizing the system resources available to you, and ensuring that the service can scale to meet traffic requirements.

V8 is a single thread JavaScript engine. This is good enough for the Chrome browser but it means a Node-based server on that shiny new 16 core server will have 1 CPU core going flat out, and 15 CPU cores sitting idle. Your manager may want an explanation for this.

A single thread process will only use one core. That's a fact of life. Using multiple cores in a single process requires multi-threaded software. But Node's *no threads* design paradigm, while keeping the programming model simple, also means that Node does not make use of multiple cores. What are you to do? Or more important, how are you to keep your manager happy?

The general technique is to share out incoming requests to several processes, or threads, depending on application server architecture choices. Bigger installations take this a step further, and use a load balancer to distribute requests to multiple servers.

Starting with Node 0.8.x, the `cluster` module in the `Node.js` core allows the programmer to create a "network of processes that all share server ports" These processes all live on the same machine, making the `cluster` module suitable for filling up the cores on a single server. It doesn't help with distributing work over multiple servers.

The `cluster` module is fairly low level, and it can be rewarding to develop your scalability on top of `cluster`. However, there are a couple of packages that simplify its use, and for Notes, we will use the `workforce` module. That module is inspired by an earlier module, also named `cluster`, which was supplanted by the one in the Node core.

To get started, add this module dependency to the `package.json` file:

```
'dependencies': {  
  ..  
  'workforce': '*'  
}
```

Run `npm install` to ensure the module is available.

Then, create a file named `workforce.js`, in the same directory as `app.js`, containing the following code:

```
var workforce = require('workforce');
var manager = workforce('./app.js')
  .set('workers', 4)
  .set('title', 'Notes')
  .set('restart threshold', '10s')
  .set('exit timeout', '5s');
manager.listen(process.env.PORT || 3000);
```

Next, in `app.js` make a couple of small changes.

```
var app = module.exports = express();
```

This makes `app.js` into a regular module, with the `app` object exported as the module object. This means other modules can manipulate the `app` object for further customization. Next, comment out the code at the end which configures the Express server object to listen.

```
//http.createServer(app).listen(app.get('port'), function() {
//  console.log('Express server listening on port '
//    + app.get('port'));
//});
```

What happens is that the `workforce` module takes care of configuring the `app` object to listen to the correct port. Exporting the `app` object, by assigning it to `module.exports`, gives the `workforce` module the ability to give each worker process the configuration necessary for all to work together.

In `workforce.js`, you can easily configure the number of worker processes that are launched by changing this line:

```
.set('workers', 4)
```

The value to use here depends on your system configuration, but should probably be equal to the number of cores or processors on the server.

Launching the server is a little different:

```
$ node workforce.js
```

Next you can verify that there are multiple worker processes:

```
$ ps -eaf | grep node
5708 ttys001 0:03.76 node workforce.js
5709 ttys001 0:00.58 node ../worker.js Notes app.js 3000
```

```
5710 ttys001 0:00.58 node ../worker.js Notes app.js 3000
5711 ttys001 0:00.58 node ../worker.js Notes app.js 3000
5712 ttys001 0:00.58 node ../worker.js Notes app.js 3000
5745 ttys002 0:00.00 grep node
```

We use the application in exactly the same way as we did before, however it will now handle a larger traffic load. So tell all your friends and see what happens.

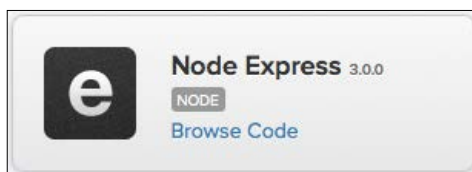
Deploying Notes on cloud hosting (AppFog)

With the `workforce` module we were able to scale Notes to use up all the cores on the server. This handles scaling within one server, but to scale your website beyond one server requires configuring a multi-server system. Cloud hosting systems promise the ability to automatically scale the number of servers as traffic increases or decreases throughout the day.

It can be really complex to set up a cloud hosting system. A few cloud hosting companies promise to simplify away cloud hosting complications, some of whom offer Node cloud hosting.

Let's look at this by using AppFog to host the Notes application.

To start, go to <http://appfog.com> and register for a free account. Once you're registered, go to the dashboard and click on the **Create App** button. On the page that follows, click this button:



This sets up a blank minimized Node application. The source code for that app is on GitHub at <https://github.com/appfog/af-node-express>, should you want to take a look. You'll also need to choose a deployment region, and a domain name it will be deployed on.



The deployment region refers to a range of cloud hosting infrastructures into which AppFog can deploy applications. The domain name you chose also acts as the application name. By default the actual domain name will be a subdomain of AppFog's own domain, but you can attach a custom domain name if desired.

This sets up the application, and gives you an application dashboard. For example, you can control the number of instances and allowed memory footprint with sliders on the dashboard page. The dashboard supports a variety of configuration options and facilities, which can be attached to the application, such as environment variables, domain names, and SSL.

For the Notes application we need a database to store the notes. Out of the box, AppFog makes it easy to set up MongoDB, PostgreSQL, MySQL, RabbitMQ, or Redis services. This is as simple as clicking on the **MySQL** button, typing in a service name, and clicking on the **Create** button. Once the database is created, make sure the database service is bound to the application.

Each item we deal with within the AppFog dashboard has a simple text name. The application has its own name, as does each service. It's helpful to choose names that make it clear which is which, because we'll be using these names in code later on.

Services
For more information on how to connect your app to services, check out [our docs](#)
When you bind services to your app, it modifies an environment variable called `VCAP_SERVICES`.

Bound Service	Description	Service	
 notes-db	database	mysql 5.1	

You have 0 services on other infrastructures.

As the screen says, configuration data about the bound services is available in the `VCAP_SERVICES` environment variable. This variable will contain a JSON object that you're expected to parse. That data includes the database connection credentials required to access the database.

In `app.js`, then, we have to replace the hard-coded object for database credentials, with code that grabs the credential information from `VCAP_SERVICES`. This is a really positive step that helps our program by removing hard-coded configuration details.

Because AppFog handles scaling the number of instances, we do not require something such as the `cluster` or `workforce` modules. That means we'll be ignoring the `workforce.js` module, and launching the application using `app.js` again. The `app.js` module will require a few modifications.

The first modification to get out of the way is at the bottom of the file, where we set it up to listen for HTTP connections. With the `workforce` module, we commented this out from `app.js`, but for deployment on AppFog, the application is told what port to listen on through the `VCAP_APP_PORT` environment variable. Change the code at the end of `app.js` to read as follows:

```
// Comment this out to run with workforce or on appfog
//http.createServer(app).listen(app.get('port'), function() {
//  console.log('Express server listening on port '
//    + app.get('port'));
//});
// Use this line for appfog
app.listen(process.env.VCAP_APP_PORT || 3000);
```

The next thing to change is the database configuration for the data models. Instead of hardcoding the database configuration in `app.js`, we'll extract it from the `VCAP_SERVICES` environment variable.

In `app.js`, insert this code before setting up the data models:

```
var seqConnectParams = {};
if (process.env.VCAP_SERVICES) {
  var VCAP = JSON.parse(process.env.VCAP_SERVICES);
  for (var svctype in VCAP) {
    if (svctype.match(/^mysql/)) {
      VCAP[svctype].forEach(function(svc) {
        if (svc.name === 'notes-db') {
          seqConnectParams.dbname = svc.credentials.name;
          seqConnectParams.username = svc.credentials.username;
          seqConnectParams.password = svc.credentials.password;
          seqConnectParams.params = {
            host: svc.credentials.host,
            dialect: 'mysql'
          };
        }
      });
    }
  }
} else {
```

```
// seqConnectParams = './chap06.sqlite3';
// seqConnectParams = 'mongodb://localhost/chap06';
seqConnectParams = {
  dbname: '...', username: '...', password: '...',
  params: {
    host: '...',
    dialect: 'mysql'
  }
};
}
```

This code looks for the `VCAP_SERVICES` environment variable, and if present, extracts it from the database configuration. It looks for MySQL services, within which it expects to find a service named `notes-db`. You may end up with a different service name in AppFog, so modify this code to match the service name you are using.

The `VCAP_SERVICES` variable contains configuration data for the services that are configured for the application. Some of those services, like this one, are databases. You'll note from our code that we retrieve the actual database name from the `VCAP_SERVICES` variable entry for the service.

If the environment variable is not found, it defaults to the hard-coded database configuration. If you're using MongoDB instead, change the code to suit.

Next, change how the data models are configured:

```
var notesModel = require('./models-sequelize/notes');
notesModel.connect(seqConnectParams,
  function(err) {
    if (err) throw err;
  });
[ routes, notes ].forEach(function(router) {
  router.configure({ model: notesModel });
});
usersModel.connect(seqConnectParams,
  function(err) {
    if (err) throw err;
  });
users.configure({
  users: usersModel,
  passport: passport
});
```

The change here is that instead of the hard-coded database configuration, we use the one extracted from the environment variable.

Now we must upload the source code to the AppFog service. To do so, we'll use the AppFog command-line tool. That tool has many options, and does many more things than you can do through the AppFog website.

Install the AppFog command-line tool:

```
$ sudo gem install af
```

Of course, if you're using Windows, the `sudo` portion of the command line is unneeded. You must have Ruby installed as well. One way to do so is by using the installer available at RubyInstaller.org. You may need to install a 32-bit version of Ruby with a version number less than 2.0.0, if the following commands fail for you.

Use it to upload your source code.

```
$ af login
Attempting login to [https://api.appfog.com]
Email: me@example.com
Password: *****
Successfully logged into [https://api.appfog.com]
$ af update notes-nwb
Uploading Application:
  Checking for available resources: OK
  Processing resources: OK
  Packing application: OK
  Uploading (93K): OK
```

The `af update` command updates an application, and in this case, the application name is `notes-nwb`. The name you give here must be the name given when creating the application, and is shown in the AppFog dashboard.

If all has gone well, after the `af update` command stops printing messages, you can visit your application and be greeted by the familiar Notes application, except there are no Notes in the database. Furthermore, if you try to create any Notes, the application kicks you over to the login screen (as expected), because there are no users in the database and you cannot log in.

In short, it's necessary to set up a couple of user table entries so that we can log in and begin creating Notes.

To do this, we will use the AppFog command-line tool (`af`), to connect with the database, and using the `setup.js` tool we used earlier, initialize our user table entries.

The `af` command-line tool can create a **tunnel** to services running on AppFog's servers. Through that tunnel we can run MySQL commands and initialize our database.

This is how we use the tool to create the tunnel:

```
$ af tunnel
1: notes-db
Which service to tunnel to?: 1
Getting tunnel connection info: OK
Service connection info:
  username : ..
  password : ..
  name      : ..
  infra     : aws
Starting tunnel to notes-db on port 10000.
1: none
2: mysql
3: mysqldump
Which client would you like to start?: 1
Open another shell to run command-line clients or
use a UI tool to connect using the displayed information.
Press Ctrl-C to exit...
```

The username, password, and database name credentials printed by `af` are required to log in to the MySQL server. Instead of the normal MySQL port, the tunnel is open on port 10000, allowing you to do this, in a new command window:

```
$ mysql --port=10000 -user=.. --password=.. dbname
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
```

The values given for `-user=`, `--password=`, and `dbname` are the same as printed earlier. Using the `mysql` command tool you can browse the database and run SQL commands. While that's cool and useful, what we're looking to do is use `setup.js` to initialize the user table.

Change `models-sequelize/setup.js` as so:

```
var users = require('./users');
users.connect({
  dbname: '..', username: '..', password: '..',
  params: {
    host: 'localhost',
    port: 10000,
    dialect: 'mysql'
  }
},
function(err) {
  if (err) throw err;
  else {
    users.create('1', 'bob', 'secret',
      'bob@example.com', function(err) {
        if (err) throw err;
        else {
          users.create('2', 'joe', 'birthday',
            'joe@example.com', function(err) {
              if (err) throw err;
            });
        }
      })
  }
});
```

This is the same as as we used before, but with a different connection object. The database connection parameters to use are the ones printed by the `af tunnel` command. The `af tunnel` command printed the database name, user name, and password, so simply plug those values into the connection object.

Once you set this up, run the `setup.js` script like so:

```
$ node models-sequelize/setup.js
```

Because this script is run locally, there's no need to upload it to AppFog.

Now that we have user entries in the `users` table, we can log in and begin entering notes in the application.

Summary

We've come a long way in this chapter about user authentication in Node, and several aspects of deploying and scaling Node applications.

Specifically we covered:

- Session management in Connect and Express
- Cookie handling in Express
- The passport user authentication system
- Router middleware in Express
- Setting up multiple-data models
- The cluster and workforce modules to scale within the server
- Using forever to keep a background Node application running
- Configuring an LSB-compliant script to start a Node application on Debian at system boot
- Deploying applications to AppFog
- Tunneling to the database service on AppFog

In the next chapter, we'll be taking the Notes application to a whole new level. We'll be exploring how to bridge between the browser side and server side, using JavaScript code on both ends to create real-time communication between the two.

8

Dynamic Interaction between the Client and Server Application

The original design model of the web is similar to the way that mainframes worked in the 1970s. The interaction pattern for both old-school dumb terminals (such as the IBM 3270) and web browsers follows a request-response paradigm. While web browsers can show more complex information than old-school dumb terminals, the interaction pattern in both cases is a back-and-forth of user requests, each resulting in a *screen* of data sent by the server. Screen after screen, or in the case of web browsers, page after page. The future is moving rapidly. Web browsers with modern JavaScript engines can do so much more than the dumb terminals of yesteryear, or even the web browsers of just a couple years ago. One new technique is to keep a connection open with the server for continual data exchange between server and client, representing a change in the web application model. Some call this the real-time web.

With JavaScript on both the server and client, we can now implement a dream that dates back to the days when Java applets tried to convince us that the browser could host interactive stuff on web pages. JavaScript in the client makes it possible to keep the server connection open for exchanging data with the server. Having JavaScript on both the server and client means more facile and fluid data sharing between the frontend and backend of the application.

One observation we can make is that traditional web applications can untruthfully display their data. That is, if two people are looking at a page, such as a wiki page, and one person edits that page, that person's browser will update with the correct copy of the page, while the other browser will not update. The two browsers are showing different versions of the page, one of which is untruthful. Some think it would be better if the other person's browser refreshed to show the new content as soon as the page is edited.

This is one possible role of the real-time web; pages that update themselves as the page content changes. We're about to implement this behavior in the Notes application.

The more common role of the real-time web is the way some of the social networking websites now dynamically update the page as other people make posts or comments, or pop up activity notifications as other people within the social network make comments or new posts. We're about to implement a tiny messaging system inside the Notes application, as well.

Instead of a page-by-page interaction between the browser and the server, the real-time web maintains a continual stream of data going between the two, with application code living in the browser, to implement the browser-side behavior.

One of the original purposes for inventing Node in the first place was to support real-time web implementation. The Comet application architecture involves holding the HTTP connection open for a long time, with data flowing back and forth between the browser and server over that channel. The term Comet was introduced by Alex Russell on his blog in 2006 (<http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/>), as a general term for the architectural pattern to implement this real-time, two-way data exchange between the client and server. That blog post called for development of a programming platform very similar to Node.

Adding real-time web features to Notes

To explore this, we're going to add some real-time web functionality to the Notes application.

To simplify the task, we'll be leaning on the Socket.IO library. This library simplifies two-way communication between the browser and server and can support a variety of protocols with fallback all the way to Internet Explorer 5.5.

A common companion to Socket.IO is Backbone.js, because it supports data models, allowing the programmer to exchange high-level data over Socket.IO connections. Because the data in the Notes application is so simple, we won't use Backbone. It is, however, worth exploring (<http://backbonejs.org/>).

Another pair of packages worth exploring are DerbyJS (<http://derbyjs.com/>) and ShareJS (<http://sharejs.org/>). The latter supports real-time editing between multiple users (it was written by a former Google Wave engineer), while DerbyJS makes it easy to build real-time collaborative applications.

Introducing Socket.IO

The aim of Socket.IO is:

To make realtime apps possible in every browser and mobile device.

It supports several transport protocols, choosing the best one for the specific browser.

If you were to implement your application with WebSocket, it would be limited to the modern browsers supporting that protocol. Because Socket.IO falls back on so many alternate protocols (WebSockets, Flash, XHR, and JSONP) it supports a wide range of web browsers, including Internet Explorer 5.5.

As the application author, you don't have to worry about the specific protocol Socket.IO uses in a given browser. Instead, you implement the business logic, and the library takes care of the details for you.

Socket.IO requires that a client library make its way into the browser. To make that easy, the Socket.IO server listens for a request on a given path, which by default is `/socket.io/socket.io.js`, responding by sending the client library to the browser.

The model Socket.IO provides you with is akin to the EventEmitter object. The programmer uses the `.on` method to listen for events, and the `.emit` method to send them. The emitted events are sent between the browser and the server, with the Socket.IO library taking care of sending them back and forth.

Initializing Socket.IO with Express

Socket.IO works by wrapping itself either around an HTTPServer object, or around an Express application object. To understand that, think back to *Chapter 4, HTTP Servers and Clients – A Web Application's First Steps*, where we used the HTTP Sniffer to view the events emitted by the HTTPServer object. The HTTP Sniffer attaches a listener to every HTTP event, to print out the events. But what if you used that idea to do real work? Socket.IO uses a similar concept, listening to HTTP requests, and responding to specific ones to use the Socket.IO protocol to communicate with client code in the browser.

Adding the Socket.IO library to an Express application requires just a few simple changes, which we're about to look at.

To get started, make a duplicate of the Notes application used in *Chapter 7, Multiuser Authorization, Deployment, Scaling, and Hosting*, and then follow along with these changes.

In `package.json` add this dependency:

```
"socket.io": "~0.9.x"
```

This brings in the Socket.IO library version we're using in this chapter. It's known that, sometime in the future, Socket.IO 1.0 will be released and requires some changes. What's not known is when that will occur.

In `app.js`, the HTTP server gets initialized a little differently for use with Socket.IO, and the Socket.IO library attaches itself to the HTTP server.

The first step is as always:

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

Then, following the configuration and routes setup, add these lines:

```
var server = http.Server(app);  
var io = require('socket.io').listen(server); // socket.io 0.9.x
```

This is where `socket.io` wraps itself around the server. It adds its own event handlers so that it can handle HTTP requests related to the Socket.IO protocol.

Then at the very end of the file use this:

```
server.listen(app.get('port'), function(){  
  console.log("Express server listening on port "  
    + app.get('port'));  
});
```

There is another section of code to add at the end of `app.js`. This section will contain the application business logic code for sending appropriate events, and communicating data, to and from the browser.

Socket.IO's model is the EventEmitter class, extending its API, as we said earlier, for the methods that send events between the browser and server. The implementation of this is event handler functions in both the server and the browser code.

The container for the server-side handler functions will be placed at the end of `app.js` inside this wrapper:

```
io.sockets.on('connection', function (socket) {  
  ..  
});
```

This sets up an event handler listening for web browsers to connect to the server. Within this container we'll set up event handler functions, on the `socket` object, to receive events from the browser. Socket.IO maintains an open connection between them, with the `socket` object being our hook into that open connection. It is also an EventEmitter and to listen for an event we use the `.on` function to set up an event listener, and to send an event we use the `.emit` function.

We'll go over this in more detail in a few pages.

Setting up the client code

As we said earlier, Socket.IO requires client code in the browser. The steps are to first load the client-side Socket.IO library into the page, then to connect with the server, and finally to set up event handler functions to interact with the server. The client-side functions are to be the mirror of the server-side functions. Each side sends the events, which the other side catches.

Edit `views/top.ejs` and change the `<head>` section to read as so:

```
<head>
  <title><%= title %></title>
  <script src="http://code.jquery.com/jquery-1.9.1.min.js">
  </script>
  <link rel='stylesheet' href='/stylesheets/style.css' />
  <script src="/socket.io/socket.io.js"></script>
  <script>
    var socket = io.connect('/'); // socket.io-client 0.9.x

  </script>
</head>
```

Loading the code into the page is done by retrieving it from the special path, `/socket.io/socket.io.js`. This is recognized by the server-side Socket.IO library, and it responds by sending the client-side library. It's a very simple way to bootstrap Socket.IO into the browser.

The next step is to use the client-side library to connect with the server side, using the `connect` function. The code required to initialize this connection is expected to change in future Socket.IO versions.

We have not shown the event handler functions for this page, but will look at their implementation in a later section.

Events between the Notes server and client code

Socket.IO extends the EventEmitter concept to allow communication, across the network, between the server and client. Before implementing our code, we must know which events our application will require for communicating transactions between the server and client. The exact events for a given application of course are determined by the application needs.

The real-time functions we'll add to Notes will be in the main page, and in the `noteview` page. In both cases, there are opportunities for someone else to add, modify, or delete a note which in turn means updating other clients of the change.

With that in mind, ponder over these events:

- `notetitles`: This event is sent by the Notes home page, from the browser, when it wants to refresh the list of notes. The home page is to send this event when it wants to update itself with the list of note titles. This way as Notes are created, updated, or deleted, the home page can update itself.
- `noteupdated`: This event is sent by the server whenever a given note is updated, such as someone editing and saving a note. This event is used by both the Notes home page, and the note view page, to update the display when the content changes.
- `notedeleted`: This event is sent by the server whenever a given note is deleted. On the Notes home page, this event is used to refresh the list of notes. For someone viewing a note that is deleted, it is no longer appropriate for them to continue viewing that note, and their browser will be redirected to the Notes home page.

This is all that is required for the Notes application. It is that simple.

When thinking about the events your application needs to send, keep in mind that there will be multiple users simultaneously using the application. There will be a constant back-and-forth of activity between their browsers and the server. The nature of that activity depends on the transactions in your application.

Each section of your application has something akin to a finite state machine; as a state transition occurs, it is accompanied by the sending and receiving events.

The purpose of all this is to distribute knowledge of what's happening between application users, as it happens.

Now, let's see how to implement these events.

Modifying the Notes model to send events

The events to be sent by the Notes application, `noteupdated` and `notedeleted`, are best sent by the model code. It is the models that know when a note is modified or deleted. That's because, like many kinds of applications, any transaction undertaken on behalf of the user ends up as a transaction in the model. In this chapter, we'll work solely with the Sequelize model, but similar changes can easily be made in the other models.

What we're going to do is have the model emit events, using an `EventEmitter`, as a by-product of the actions taken by the Model.

In `models-sequelize/notes.js`, add the following code:

```
var events = require('events');
var emitter = module.exports.emitter = new events.EventEmitter();
```

The model is exporting an `EventEmitter` that we'll use to send the events. Any code that is interested in an event should call `notesModel.emitter.on`, and when it loses interest, it should call `notesModel.emitter.removeListener`.

We'll add calls to `emitter.emit` when needed to send the events.

Change the `create` function to add this:

```
exports.create = function(key, title, body, callback) {
  Note.create({
    notekey: key,
    title: title,
    body: body
  }).success(function(note) {
    exports.emitter.emit('noteupdated', {
      key: key, title: title, body: body
    });
    callback();
  }).error(function(err) {
    callback(err);
  });
}
```

This sends out the `noteupdated` event along with its data. Note that we don't send the `note` object, because each model has its own internal representation in which it is storing the note. In this case, it is a Sequelize object that has other fields beyond what has been defined for a note. It's best to send a sanitized object, as we're doing.

In the update function, we'll make a similar change:

```
exports.update = function(key, title, body, callback) {
  Note.find({ where: { notekey: key }}).success(function(note) {
    if (!note) {
      callback(new Error("No note found " + key));
    } else {
      note.updateAttributes({
        title: title,
        body: body
      }).success(function() {
        exports.emitter.emit('noteupdated', {
          key: key, title: title, body: body
        });
        callback();
      }).error(function(err) {
        callback(err);
      });
    }
  }).error(function(err) {
    callback(err);
  });
}
```

Likewise, this sends out the noteupdated event along with its data.

Then in the destroy function, make this change:

```
exports.destroy = function(key, callback) {
  Note.find({ where: { notekey: key }}).success(function(note) {
    if (note)
      note.destroy().success(function() {
        emitter.emit('notedeleted', key);
        callback();
      }).error(function(err) {
        callback(err);
      });
    else callback();
  });
}
```

This sends out the notedeleted event, along with the key for the note that was just deleted.

Notice how in each case we're sending just the relevant information in the event, and no more. One reason for this is that when the application broadcasts a message to all browsers, it's best to keep the message small to minimize the network bandwidth consumed by the application.

Socket.IO lets you send JavaScript objects back and forth between server and client, and it takes care of encoding everything correctly.

Let's now see how to connect these events with the server side of the communication with the browser.

Sending the events from the Notes server

Now we can start to look at how to implement the communication between the server and the client. We add this block of code to the end of `app.js`:

```
io.sockets.on('connection', function (socket) {
  socket.on('notetitles', function(fn) {
    notesModel.titles(function(err, titles) {
      if (err) {
        util.log(err);
        // send error message ??
      } else {
        fn(titles);
      }
    });
  });
});

var broadcastUpdated = function(newnote) {
  socket.emit('noteupdated', newnote);
}
notesModel.emitter.on('noteupdated', broadcastUpdated);
socket.on('disconnect', function() {
  notesModel.emitter.removeListener('noteupdated',
    broadcastUpdated);
});

var broadcastDeleted = function(notekey) {
  socket.emit('notedeleted', notekey);
}
notesModel.emitter.on('notedeleted', broadcastDeleted);
socket.on('disconnect', function() {
  notesModel.emitter.removeListener('notedeleted',
    broadcastDeleted);
});
});
```

This is the code we mentioned earlier, which implements the server side of the three events we discussed earlier. The main function is to listen for events occurring when browsers connect to the server.

The first part responds to a `notestitle` request sent by the browser code. The server response is to request the note titles array from the model, sending it to the browser. We'll see, in a later section, how that array gets rendered on the page.

Notice that we set up this handler using `socket.on`. This is the same `.on` function we know from the `EventEmitter` class. The `socket` object is an `EventEmitter`, which sends several events, such as the `connect` and `disconnect` events we're listening to here.

The next two sections of code listen to the `noteupdated` and `notedeleted` events emitted by the data model. In each case, the event handler simply uses `socket.emit` to send that data to the browser. There's nothing to do other than relay the event out to the browser.

The other thing we do is to listen for the `disconnect` event. This is sent when the page disconnects from the server, such as when the browser visits another page. When this event is sent, we must disconnect the listener function from the data model using the `removeListener` function. If we didn't do this, there would be a build-up of listener functions as pages connect to the server, but nothing is done when the page disconnects from the server.

Browser-side event handlers

Now we can look at how to implement the browser-side of the code we've just looked at.

Events sent by the server appear in the browser, thanks to the `Socket.IO` client library, and likewise the browser can send events to the server, thanks to this library.

Earlier, we made sure each page loads the `Socket.IO` client library into the browser, using a script tag, and then initializing that library connects to the server. What's left is to implement the per-page business logic of the application.

Each page in the Notes application has different browser-side code, because the actions to undertake differ on each page. What we're now going to do is implement the browser-side code.

In `views/index.ejs`, use this code:

```
<% include top %>

<div id='notetitles'>
<%
```

```

    if (notes) {
      notes.forEach(function(note) {
        %><p><%= note.key %>:
        <a href="/noteview?key=<%= note.key %>">
          <%= note.title %></a>
        </p><%
      });
    }
    %>
  </div>

  <script>
    $(document).ready(function() {
      var getnotetitles = function() {
        socket.emit('notetitles', function (notes) {
          $('#notetitles').empty();
          notes.forEach(function(note) {
            $('#notetitles').append('<p>' + note.key
              + ': <a href="/noteview?key='+ note.key +' ">'
              + note.title + '</a></p>');
          });
        });
      };
      socket.on('noteupdated', getnotetitles);
      socket.on('notedeleted', getnotetitles);
    });
  </script>

  <% include bottom %>

```

This is similar to the previous version of `index.ejs`.

The first part is what we had before, a loop to render data on the server side. This time we are wrapping that code with a `<div id='notetitles'></div>` block, so we can use some jQuery magic to replace the content when we have updates.

The second part is the browser-side Socket.IO code, implemented with a little help from jQuery. Remember that, on this page, the task is to update the list of Notes any time a note is changed or deleted.

We wrap this inside `$(document).ready`, to ensure the execution occurs after the page is fully loaded.

We are using `socket.on` to listen to two events, `noteupdated` and `notedeleted`, then for each to send a `notetitles` event.

To update the page, we use some jQuery code to simplify the DOM manipulation. It takes two steps; the first is to delete the existing content of the `notetitles` block, and then secondly to add in the new content.

This page demonstrates the advantage of JavaScript on both the client and server. The JavaScript code that's surrounded by the `<%` and `%>` markers is executed by the server, while the JavaScript code appears inside a `<script>` tag, and is executed by the browser. It's all JavaScript at both ends of the wire.

We have to be careful writing this, remembering carefully which context is executing each block of JavaScript. It means our heads don't have to switch gears between languages to write both ends of the application. But we do have to switch gears over the capabilities of the browser and server.

In `views/noteview.ejs`, use this code:

```
<% include top %>
<div id="noteview">
  <h3 id="notetitle"><%= note ? note.title : "" %></h3>
  <p id="notebody"><%- note ? note.body : "" %></p>
  <p>Key: <%= notekey %></p>
  <% if (user && notekey) { %>
    <hr/>
    <p><a href="/notedestroy?key=<%= notekey %>">Delete</a>
    | <a href="/noteedit?key=<%= notekey %>">Edit</a></p>
  <% } %>
</div>

<script>
  $(document).ready(function() {
    var updatenote = function(newnote) {
      $('#notetitle').empty();
      $('#notetitle').append(newnote.title);
      $('#notebody').empty();
      $('#notebody').append(newnote.body);
    }
    socket.on('noteupdated', function(newnote) {
      if (newnote.key === "<%= notekey %>") {
        updatenote(newnote);
      }
    });
    socket.on('notedeleted', function(notekey) {
      if (notekey === "<%= notekey %>") {
        document.location.href = "/";
      }
    })
  });
</script>
<% include bottom %>
```

The first part is again similar to what we've used before. We added a couple of `id` attributes to aid using jQuery for DOM manipulation. The second part is where we act on the events, and do the DOM manipulation.

The page is set up to act on two events, `noteupdated` and `notedeleted`. Both events carry the key for the affected note. The Notes application sends these events to all browsers, irrespective of whether the browser is viewing the note. However, it is only those browsers that are viewing this note that should act on the event.

A browser viewing the `noteview` page is viewing a specific note, which has an identifying key. When the page is built on the server, we are supplied the key in the `notekey` parameter. We must not act on events for Notes other than the one being displayed in the browser. The first thing to be done is to check whether the event refers to the note displayed on this page.

The `notekey` parameter is a value known on the server, but the verification that the event refers to the currently displayed note, occurs in the browser. In this case, we have to mix the browser-side JavaScript with a bit of server-side code by using `<%= notekey %>` to substitute in the value of the server-side variable.

Upon verifying that the event refers to the Note being displayed, we perform one of two actions. For the `noteupdated` event, we use the `updatenote` function to perform DOM manipulation to replace the note content being displayed.

For the `notedeleted` event, it's no longer appropriate to keep the browser on this page, because the note no longer exists. In this case, we cause the browser to redirect to the home page of the Notes application.

The behavior we've implemented for the `notedeleted` event does raise a question: Is this the best, most appropriate, user experience for that event? A real application for real users should, of course, have a well-designed user experience. Your users will appreciate any user interface clarity you design into your application.

Running the Notes application with Socket.IO

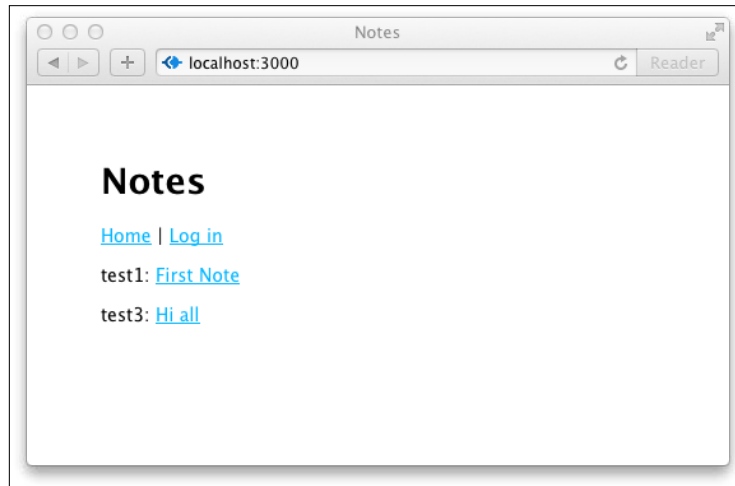
Now we're ready to run the application. First do this to ensure that the `socket.io` library is installed, because of the dependency added in `package.json` earlier.

```
$ npm install
```

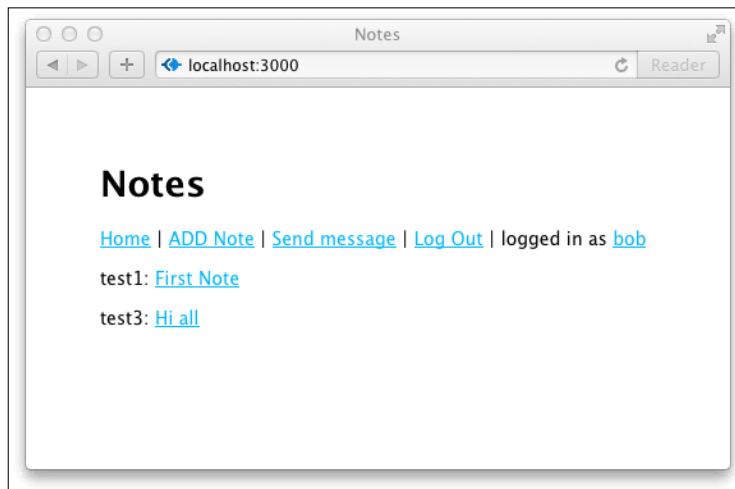
Then run the application as before:

```
$ node app.js
```

Then you will be greeted with the familiar home page:



You will be able to log in as normal:



And you can take the application through its normal paces. The new behavior shows itself when you have multiple browsers connected to the application.

When viewing the home page in one browser, watch what happens as you create, update, or delete notes in another browser. Likewise, if viewing a note in a browser, watch what happens as you edit or delete that note in another browser.

As you run the server, it will print a running log of the Socket.IO activity, letting you see what it's doing:

```
debug - client authorized
info - handshake authorized m4MA-s-U5PflPo08kMJL
debug - setting request GET /socket.io/1/websocket/m4MA-s-
U5PflPo08kMJL
debug - set heartbeat interval for client m4MA-s-U5PflPo08kMJL
debug - client authorized for
debug - websocket writing 1::
debug - emitting heartbeat for client m4MA-s-U5PflPo08kMJL
debug - websocket writing 2::
debug - set heartbeat timeout for client m4MA-s-U5PflPo08kMJL
debug - got heartbeat packet
debug - cleared heartbeat timeout for client m4MA-s-U5PflPo08kMJL
debug - set heartbeat interval for client m4MA-s-U5PflPo08kMJL
debug - emitting heartbeat for client m4MA-s-U5PflPo08kMJL
debug - websocket writing 2::
```

If you are running with Socket.IO 1.0, you enable this logging output by setting the `DEBUG` environment variable before starting the server like so:

```
$ export DEBUG=""
$ node app.js
```

Or, in Windows, do it this way:

```
C:\> set DEBUG=""
C:\> node app.js
```

Listening to the heartbeat and cleaning up when it stops

In the logging output just shown, there are messages about a heartbeat interval, and related heartbeat activities. If you let the server run for a while it's clear that Socket.IO keeps a continuous chatter between the client and server, using that to know with certainty how many browsers are connected to the server and how to communicate with them.

Earlier we raised the question of cleaning up listeners when a browser goes away. Inside the Notes application, resources are dedicated to each browser page connected to the server, and we want to ensure to release those resources when the browser page disconnects.

To start, let's modify some of the code in `app.js` as follows:

```
..
socket.on('disconnect', function() {
  util.log('disconnect - removing noteupdated listener');
  notesModel.emitter.removeListener('noteupdated',
    broadcastUpdated);
});
..
socket.on('disconnect', function() {
  util.log('disconnect - removing notedeleted listener');
  notesModel.emitter.removeListener('notedeleted',
    broadcastDeleted);
});
..
socket.on('disconnect', function() {
  util.log('disconnect - removing newmessage & delmessage
listeners');
  usersModel.emitter.removeListener('newmessage',
    broadcastNewMessage);
  usersModel.emitter.removeListener('delmessage',
    broadcastDelMessage);
});
..
```

This adds logging messages so we can see if the server is catching the disconnect message.

Next, start up the server and connect to it with one or more browsers. Verify that the heartbeat messages are being logged, and that it has a heartbeat going for each connected browser. For example:

```
debug - emitting heartbeat for client UfsLqjmXXcqz_Q8ueRU7
..
debug - emitting heartbeat for client SMoyqzh22-3j5pI3eRU3
```

This shows two client browsers connected, each with their own code string.

The last step is to kill one of the browsers, and this should be logged to the terminal window:

```
info - transport end (socket end)
debug - set close timeout for client UfsLqjmXXcqz_Q8ueRU7
debug - cleared close timeout for client UfsLqjmXXcqz_Q8ueRU7
debug - cleared heartbeat interval for client UfsLqjmXXcqz_Q8ueRU7
28 Jun 13:37:53 - disconnect - removing noteupdated listener
28 Jun 13:37:53 - disconnect - removing notedeleted listener
28 Jun 13:37:53 - disconnect - removing newmessage & delmessage listeners
debug - discarding transport
```

The good news is that not only is the server seeing the socket getting closed, the `disconnect` event is also sent. As we wrote the code correctly, and we know that because these messages are printed, the internal event listeners are being removed.

Sending messages between users

Let's try something a little more complex, and closer to the intended purpose of Socket.IO. Let's implement a way for Notes' users to send each other messages within Notes.

When a user logs in, and they have messages, we'll show those messages in a box at the top of the screen. We'll also allow the user to delete those messages, and of course to send messages to themselves or other users.

Let's get started.

Socket.IO events for sending messages between users

The events we'll be sending and receiving are:

- `newmessage`: This event indicates that a message has been created. It includes the user ID of the recipient of the message. It's sent from the server, and the code in the browser is to retrieve the current list of messages.
- `delmessage`: This event indicates that a message has been deleted. It's sent from the server, and the code in the browser is to retrieve the current list of messages.

- `getmessages`: This is sent by the browser-side code, when it's time to retrieve the messages.
- `dodelmessage`: This is sent by the browser when the user requests to delete a message. The server will eventually respond with a `delmessage` event.

Data model to store messages

The first thing we need is a table to store the messages in. Because the messages are associated with users, we'll do this within the users model. A more complex messaging system should have its own data model. In `models-sequelize/users.js`, add this code within the `connect` function:

```
Messages = sequelize.define('Messages', {
  idTo: { type: Sequelize.INTEGER, unique: false },
  idFrom: { type: Sequelize.INTEGER, unique: false },
  message: { type: Sequelize.STRING, unique: false }
});
Messages.sync();
```

This is straightforward; each message records the user ID of the sender, the recipient, and the message that's being sent. The `unique: false` attribute is required because Sequelize, by default, wants to ensure there are no duplicated values in a column.

The users model is going to be emitting events, so add this:

```
var events = require('events');
var async = require('async');
var emitter = module.exports.emitter = new events.EventEmitter();
```

We'll be needing to retrieve a list of all the users, so that, when sending a message, we can show a selector listing the users.

```
module.exports.allUsers = function(callback) {
  User.findAll().success(function(users) {
    if (users) {
      var theusers = [];
      users.forEach(function(user) {
        theusers.push({
          id: user.id, name: user.username });
      });
      callback(null, theusers);
    } else callback();
  })
}
```

We've seen this sort of code before. The `findAll` function retrieves all instances in the user table. We simply package this up in an array and send it to the caller.

Now let's add a few functions to send these messages around:

```
module.exports.sendMessage = function(id,from,message,callback){
  Messages.create({
    idTo: id, idFrom: from, message: message
  }).success(function(user) {
    emitter.emit('newmessage', id);
    callback();
  }).error(function(err) {
    callback(err);
  });}
```

This creates a `Messages` instance in the database, and then sends a `newmessage` event.

```
module.exports.getMessages = function(id, callback) {
  Messages.findAll({
    where: { idTo: id }
  }).success(function(messages) {
    if (messages) {
      var themessages = [];
      async.eachSeries(messages,
        function(msg, done) {
          module.exports.findById(msg.idFrom,
            function(err, userFrom) {
              themessages.push({
                idTo: msg.idTo,
                idFrom: msg.idFrom,
                fromName: userFrom.username,
                message: msg.message
              });
            });
          done();
        },
        function(err) {
          if (err) callback(err);
          else callback(null, themessages);
        });
    } else callback();
  })
}
```

This function constructs the list of messages for a given user.

```
module.exports.delMessage = function(id,from,message,callback){
  Messages.find({ where: {
    idTo: id, idFrom: from, message: message
  } }).success(function(msg) {
    if (msg) {
      msg.destroy().success(function() {
        emitter.emit('delmessage');
        callback();
      }).error(function(err) {
        callback(err);
      });
    } else callback();
  })
}
```

This handles deleting a message, while sending a delmessage event.

Setting up client-side code for sending messages

Next, let's look at the HTML and JavaScript in the browser, the routes, and the routing functions.

In `app.js`, add these route declarations:

```
app.get('/sendmessage', users.ensureAuthenticated,
        users.sendMessage);
app.post('/sendmessage', users.ensureAuthenticated,
        users.doSendMessage);
```

Then in the file `routes/users.js`, add these two functions:

```
module.exports.sendMessage = function(req, res) {
  users.allUsers(function(err, theusers) {
    res.render('sendmessage', {
      title: "Send a message",
      user: req.user,
      users: theusers,
      message: req.flash('error')
    });
  });
}
```

```

module.exports.doSendMessage = function(req, res) {
  users.sendMessage(req.body.seluserid,
    req.user.id, req.body.message, function(err) {
    if (err) {
      // show error page
      res.render('showerror', {
        user: req.user ? req.user : undefined,
        title: "Could not send message",
        error: err
      });
    } else {
      res.redirect('/');
    }
  });
}

```

These route handler functions work with a new page in the application that's used for sending messages between users. The first function, `sendMessage`, renders the page on which the user enters the message. The second, `doSendMessage`, handles sending the message and displaying any error message that might be necessary.

To display this page, create a new template named `views/sendmessage.ejs`:

```

<% include top %>
<form method='POST' action='/sendmessage'>
  <p>To:
  <select name="seluserid">
    <% users.forEach(function(auser) { %>
      <option value="<%= auser.id %>">
        <%= auser.name %></option>
    <% }); %>
  </select>
</p>
  <p>Message: <input type='text' name='message' /></p>
  <p><input type="submit" value="Submit"/></p>
</form>
<% include bottom %>

```

In `sendMessage`, we retrieved the list of users, using it as the user's parameter. In HTML, we create a SELECT list to contain those users. The rest of this FORM is pretty simple, and will submit a POST request to `/sendmessage`. When we declared the routes, we said that a POST to that URL ends up in the `doSendMessage` function.

Dispatching messages between the client- and server-side

Go back to `app.js` and add the following code for connecting between the client- and server-side code.

```
io.sockets.on('connection', function (socket) {  
  ..  
  socket.on('getmessages', function(id, fn) {  
    usersModel.getMessages(id, function(err, messages) {  
      if (err) {  
        util.log('getmessages ERROR ' + err);  
        // send error??  
      } else fn(messages);  
    });  
  });  
  
  var broadcastNewMessage = function(id) {  
    socket.emit('newmessage', id);  
  }  
  usersModel.emitter.on('newmessage', broadcastNewMessage);  
  var broadcastDelMessage = function() {  
    socket.emit('delmessage');  
  }  
  usersModel.emitter.on('delmessage', broadcastDelMessage);  
  socket.on('disconnect', function() {  
    usersModel.emitter.removeListener('newmessage',  
      broadcastNewMessage);  
    usersModel.emitter.removeListener('delmessage',  
      broadcastDelMessage);  
  });  
  
  socket.on('dodelmessage', function(id, from, message, fn) {  
    usersModel.delMessage(id, from, message, function(err) {  
      // No need to do anything  
    });  
  });  
});
```

This code is similar to the glue code we wrote earlier. Its purpose is to receive messages from either the users model or the browser, passing the messages to the necessary destination.

The first receives a `getmessages` event, sent from the browser, and responds with the list of messages for the user.

The next part listens for the `newmessage` or `delmessage` events from the users model, and sends those events to the browser.

The last part receives a `dodelmessage` event from the browser, and asks the users model to delete the named message.

Displaying messages to the user

Now that we've laid the groundwork, let's see how this all works in the browser.

In `views/top.ejs`, change the initialization of Socket.IO to this:

```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = undefined;
  var delMessage = function(idTo, idFrom, message) {
    socket.emit('dodelmessage', idTo, idFrom, message);
  }
  $(document).ready(function() {
    socket = io.connect('/'); // socket.io-client 0.9.x

    <% if (user) { %>
    var getmessages = function() {
      socket.emit('getmessages', <%= user.id %>,
      function (msgs) {
        $('#messageBox').empty();
        if (msgs.length > 0) {
          msgs.forEach(function(msg) {
            $('#messageBox').append('<p>'+
              '<button onclick="delMessage('+
                msg.idTo +', '+
                msg.idFrom +', \''+
                msg.message
                  + '\'">DEL</button>'+
                msg.fromName +': '+ msg.message
              + '</p>');
          });
        }
        $('#messageBox').show();
      }
    }
    else $('#messageBox').hide();
```



```
        });  
    };  
    getmessages();  
    socket.on('newmessage', function(id) {  
        getmessages();  
    });  
    socket.on('delmessage', getmessages);  
<% } %>  
});  
</script>
```

The core of this is the `getmessages` function. It is called once when the page is loaded, and then later if a `newmessage` or `delmessage` event arrives. But first notice that everything is surrounded by this code pattern:

```
<% if (user) { %>  
..  
<% } %>
```

Remember that we have two contexts going on, one on the client side and the other on the server side. By doing this the server side is ensuring this code is sent to the browser only for logged-in users, and that a browser that isn't logged in will not have any code for sending and receiving these messages.

This function requests the list of messages for the user, by sending the `getmessages` event. We saw the implementation earlier, in that it retrieves the messages from the database and sends them back.

The messages are inserted into the `messageBox` element, which gets emptied out beforehand. If there are no messages, this element simply gets hidden. If we do have messages to display, we insert HTML to do so.

Therefore, the list of messages are refreshed any time a `newmessage` or `delmessage` event arrives.

Note that the HTML code includes a button that, when clicked, calls the `delMessage` function. That function in turn sends a `dodelmessage` message, requesting that the message be deleted. Once the server deletes the message, it'll send a `delmessage` event and as we see here, that message causes the list of messages to refresh.

The next thing to add in `views/top.ejs` is to change to the navigation bar.

```
<div class='navbar'>
<p><a href='/'>home</a>
  <% if (user) { %>
    | <a href='/noteadd'>ADD Note</a>
    | <a href="/sendmessage">Send message</a>
    | <a href="/logout">Log Out</a>
    | logged in as <a href="/account"><%= user.username
    %></a>
    <% } else { %>
    | <a href="/login">Log in</a>
    <% } %>
</p>
</div>
<% if (user) { %>
<div id="messageBox" style="display:none;"></div>
<% } %>
```

We're adding two things here. The first is the link to the `/sendmessage` URL, so that users can get to the page that's used to send messages. The second is the `messageBox` element which is where we'll display all the messages. It starts out empty, and invisible, and the JavaScript code detailed previously takes care of filling it with content and making it visible when necessary.

You may want to add some styling to the `messageBox` element so that it stands out. In `public/stylesheets/style.css` you could add something like this:

```
#messageBox {
  border: solid 2px red;
  background-color: #eeeeee;
  padding: 5px;
}
```

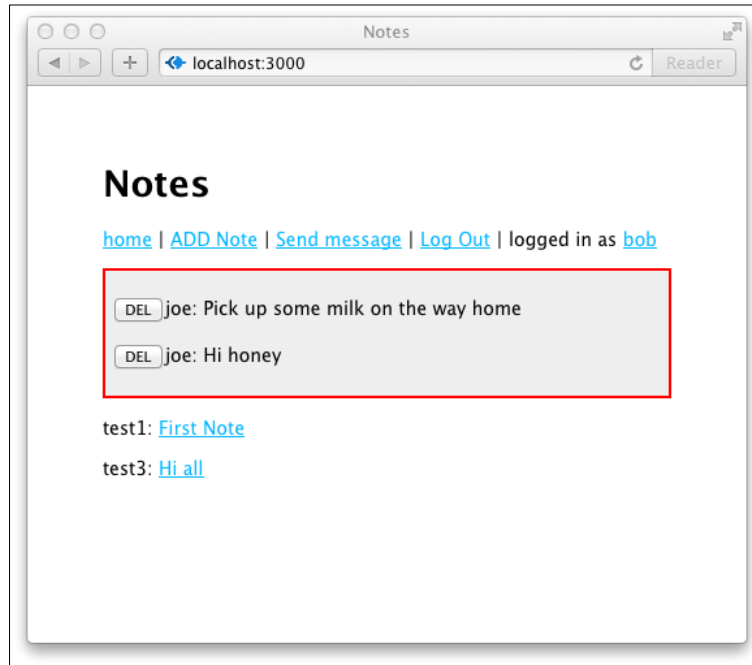
Of course there are many CSS frameworks you can use that to make this easier. We had a brief dalliance with the Twitter Bootstrap framework earlier, and it includes widgets to make this easy.

Running Notes and sending messages

Now that we have everything wired up, we can run Notes again.

```
$ node app.js
```

And, after sending a message, we'll be greeted by this:



You can test these assertions. You'll be able to delete the note by clicking the button. Click on the **Send Message** link to send a new message. Messages are displayed only for their intended recipient. The message list is automatically refreshed when a message is sent, or deleted. The message list is displayed on every page of the Notes application.

Summary

We've covered a lot of territory in this chapter, gaining an understanding of how to use Socket.IO to implement real-time communications between multiple clients connected to the same web application. With this we can now write applications that dynamically send events and data back and forth between the client and the server.

The Socket.IO library can be used to implement any kind of dynamic interaction between client and server. The server can be used as an intermediary for sending messages between users of an application or other peer-server-peer communications.

Specifically, we went over these things:

- Implementing the real-time web by holding the server-client connection open
- The Comet style protocols
- The Socket.IO library
- Using EventEmitters to send events within the server
- Writing JavaScript code in the browser
- Integrating data from the server into JavaScript on the browser
- Sending messages between the server and client using Socket.IO

In the next chapter, we'll be wrapping this book up by looking at how to test Node applications.

9

Unit Testing

Unit testing has become a primary part of good software development practices. It is a method by which individual units of source code are tested to ensure proper functioning. Each unit is theoretically the smallest testable part of the application. For Node, you could consider each module as a unit to test individually.

In unit testing, each unit is tested separately, isolating the unit from other parts of the application. Common techniques are to use mock objects, and other methods, to fake out the dependencies, so that application units can be reliably tested in isolation. We will be focusing on that testing model in this chapter.

Functional testing doesn't try to test individual components of the system, but instead tests the whole system. Generally speaking, unit testing is performed by the software development team as they write the code, and functional testing is performed by the QA or **Quality Engineering (QE)** team. Both testing models are needed to fully certify an application. An analogy might be that unit testing is similar to ensuring each word in a sentence is correctly spelled, while functional testing ensures the paragraph containing that sentence has good structure.

Now that we've written several chunks of software in this book, let's use that software to look at how to implement unit testing.

Testing asynchronous code

The asynchronous nature of the Node platform means accounting for asynchronous behavior in the tests. Fortunately, the Node unit testing frameworks help in dealing with this, but it's worth spending a few moments considering the underlying problem.

Consider a code snippet like this, which you should save in a file named `deleteFile.js`:

```
var fs = require('fs');
exports.deleteFile = function(fname, callback) {
  fs.stat(fname, function(err, stats) {
    if (err)
      callback(new Error('the file '+fname+' does not exist'));
    else {
      fs.unlink(fname, function(err) {
        if (err)
          callback(new Error('Could not delete '+fname));
        else callback();
      });
    }
  });
}
```

The nature of the asynchronous code is that the execution order is nonlinear, even though the code is written linearly, meaning there is a complex relationship between time and the lines of code. This snippet, like many written with Node, has a couple of layers of callback, making the result arrive in an indeterminate amount of time in the future.

Testing the `deleteFile` function could be accomplished with this code, which you should save in `test-deleteFile.js`:

```
var df = require('./deleteFile');
df.deleteFile("no-such-file", function(err) {
  // the test passes if err is an Error with the message
  //      "the file no-such-file does not exist"
  // otherwise the test fails
});
```

This is what we call a negative test, to verify that the failure path actually executes. It's relatively easy to verify which path was taken within the `deleteFile` function, by inspecting the `err` variable. But what if the callback is never called? That would also be a failure of the `deleteFile` function, but how do you detect that condition? You might be asking, how could it be that the callback is never called? The behavior of file access over NFS includes conditions where the NFS server is wedged and the file system requests never finish, in which case the callbacks shown here would never be called.

Another example where a callback never gets called occurs later in this chapter. In earlier iterations of the Notes application we neglected to write code to handle some errors. Our ad hoc testing did not catch the bug, and only by writing the tests cases was the bug caught.

Assert – the simplest testing methodology

Node has a useful testing tool built in the `assert` module. It's not any kind of testing framework, but simply a tool that can be used to write test code by making assertions of things which must be true or false at a given location.

The `assert` module has several methods which provide various ways of asserting conditions that must be true if the code is properly functioning.

The following is an example of using `assert` for testing, which you should put in the file `test-deleteFile.js`:

```
var fs = require('fs');
var assert = require('assert');
var df = require('./deleteFile');
df.deleteFile("no-such-file", function(err) {
  assert.throws(
    function() { if (err) throw err; },
    function(error) {
      if ((error instanceof Error)
        && /does not exist/.test(error)) {
        return true;
      } else return false;
    },
    "unexpected error"
  );
});
```

If you were looking for a quick and dirty way to test, the `assert` module could be useful when used in this way. If it runs, and no messages are printed, then the test suite passes.

```
$ node test-deleteFile.js
```

The `assert` module is used by many of the test frameworks as a core tool for writing test cases. What the test frameworks do is create the familiar test suite and test case structuring to test code.

The Vows test framework, which we're using in this chapter, is an example of such a testing framework.

Testing a model

Let's start our journey of using the Notes application to explore testing with the data models. Because this is unit testing, the models should be tested separately from the rest of the Notes application.

That would, in the case of the models, mean creating a mock database object. To test a data model and not mock out the database means that for the test to run, the database server must be running, making it a dependency of the test. However, in this case it would mean creating a fake Sequelize library, which does not look like a productive use of our time. One can also argue that testing a data model really means you're testing the interaction between your code and the database, and that mocking out the database means not testing that interaction.

Therefore we are going to skip creating a mock database module, and test the model against a real database server. Instead of running the test against the live production database, it should be run against a test database that contains test data.

Before getting started with writing test code, there are a couple tasks required for setup. The very first of which is to duplicate the Notes application used in *Chapter 8, Dynamic Interaction Between the Client and Server Application* and creating a directory named `test`, which will hold all test-related code.

```
$ mkdir test
```

Add this to the `package.json` file:

```
    "devDependencies": {  
      "vows": "*"   
    }
```

We will use the Vows framework (vowsjs.org) for writing our tests. This framework supports the behavior-driven development style of software development, meaning that it's based on **test-driven development (TDD)** principles. More importantly, it supports testing asynchronous code as part of its core competency.

In the test-driven development paradigm, our first step before writing the application code implementing a new feature is to write tests for the behavior we want. That gives us a failing test case, and theoretically we know the feature is complete when the test cases succeed.

Unlike some other test frameworks, a test script written with Vows is a regular Node script and is run with the `node` command.

To get the Vows test framework in our local workspace, run this command:

```
$ npm install
```

To aid with testability, we need to make a change in `app.js` to move the database configuration to a separate file.

Create a file named `sequelize-params.js` containing the following code:

```
module.exports = {
  dbname: "database-name",
  username: "db-user", password: "db-password",
  params: {
    host: 'db-host-name',
    dialect: 'mysql'
  }
};
```

Node modules can be used for data storage, not just functions, as shown here. This technique makes a Node module into a configuration file rather than as a container for functions. Remember that the `exports` variable becomes the object exported by the module. When it is required in another module, it is simply an object, and you can access its fields like any object.

Now, edit `app.js` to read in this configuration as so:

```
var seqConnectParams = require('./sequelize-params');
```

Having done this we can now use this piece of configuration in multiple places, without having to duplicate the configuration details in each place.

In the `test` directory, create a file named `test-model.js` containing the following:

```
var assert = require("assert")
var vows   = require('vows');
var async  = require('async');
var seqConnectParams = require('../sequelize-params-test');
var model = undefined;
var setup = function(cb) {
  model = require('../models-sequelize/notes');
  model.connect(seqConnectParams, function(err) {
    var setup = require('./setup-model');
    setup.config(model, seqConnectParams, cb);
  });
}
```

This first part sets up the test script. Notice that we are accessing `sequelize-params-test` for the database connection parameters. It is often useful to have one Sequelize database for testing, and another database for production deployment. In this case we're using a module, `setup-model.js`, which is shown in a minute, that erases existing database entries and adds entries to be used for test data. Thus, you should create a new file, `sequelize-params-test.js`, containing connection parameters necessary for a test database.

The setup brings in two modules, the Sequelize version of the Model, and a new module named `setup-model`, which we'll discuss shortly.

Now let's look at the test code which should be added to `test/test-model.js`:

```
setup(function(err) {
  vows.describe("models test")
    .addBatch({
      "check titles": {
        topic: function() { model.titles(this.callback); },
        "should have three entries": function(titles) {
          assert.equal(titles.length, 3);
        },
        "should have keys n1 n2 n3": function(titles) {
          titles.forEach(function(entry) {
            assert.match(entry.key, /n[0-9]/);
          });
        },
        "should have titles Node #": function(titles) {
          titles.forEach(function(entry) {
            assert.match(entry.title, /Note [0-9]/);
          });
        }
      },
      "read note": {
        topic: function() {
          model.read("n1", this.callback);
        },
        "should have proper note": function(note) {
          assert.equal(note.notekey, "n1");
          assert.equal(note.title, "Note 1");
          assert.equal(note.body, "Note 1");
        }
      },
      "change note": {
        topic: function() {
          model.update("n1", "Note 1 title changed",
```

```

        "Note 1 body changed", this.callback);
    },
    "after a successful model.update": {
      topic: function(err) {
        model.read("n1", this.callback);
      },
      "should be changed": function(note) {
        assert.equal(note.notekey, "n1");
        assert.equal(note.title,
          "Note 1 title changed");
        assert.equal(note.body,
          "Note 1 body changed");
      }
    }
  }
}
})
.run();
});

```

The `vows.describe` function creates a test suite. The object that's returned supports chained invocation as we've done here. Test suites contain batches, and each test batch contains what Vows calls a **context**, that contains the test cases.

The test code shown previously contains one batch, with four contexts.

The Vows framework uses JavaScript objects with verbose descriptive keys. Those descriptive keys are used by Vows to create nicely written descriptive reports of test execution results.

Each context is an object containing two sorts of things. The field named `topic` contains a function that sets up the test conditions, while all other fields contain functions that act as test cases. The latter are called `vows`, and each test context can have any number of `vows`. The `topic` field can be simply an object, or it can be a function as we've done here.

This approach separates the test code and the code being tested.

One thing we have done here is surrounded the test suite with a function named `setup`. Many test frameworks have methods that run before, or after, the test suite execution, that are meant to do the setup and tear-down steps. Vows doesn't have this capability, so we created the `setup` function to set up the database for the test suite. If you also need to do teardown, such as shutting down any services started to run the tests, Vows lets you pass a callback to the `run` function. This callback is called when all the tests are finished, so simply perform teardown in that callback.

For asynchronous functions, like the ones in our data models, Vows provides a function available as `this.callback` that captures all the arguments, passing it through Vows and into the test functions. For example, with the `read note` test, the `model.read` function is told to asynchronously call `this.callback`. Vows picks up the `Note` object passed by that function, and our test functions check various things about the note.

You'll see that these test functions have hard-coded the values each checks for. We can't do this if we don't know ahead of time the values in the database. We make this work by preconfiguring the database using the `test/setup-model.js` script. Create that file with this content:

```
var async = require('async');
exports.config = function(model, params, cb) {
  async.series([
    function(done) {
      model.connect(params, done);
    },
    function(done) {
      var delFirst = function() {
        model.titles(function(err, notes) {
          if (err) done(err);
          else {
            if (notes.length >= 1) {
              model.delete(notes[0].key,
                function(err) {
                  if (err) done(err);
                  else delFirst();
                });
            } else done();
          }
        });
      };
      delFirst();
    },
    function(done) {
      model.create("n1", "Note 1", "Note 1", done);
    },
    function(done) {
      model.create("n2", "Note 2", "Note 2", done);
    },
    function(done) {
      model.create("n3", "Note 3", "Note 3", done);
    }
  ], cb);
};
```

```

    ],
    function(err) {
      if (err) throw err;
      else {
        model.disconnect(function() { });
        cb();
      }
    });
  }
}

```

This script connects with the database model that it's given, deletes all existing data in the database, then adds a few notes that have known values. This way we have a database with known values in it, and can write a test based on the foreknowledge of the notes in the database.

Executing the tests

To execute these tests, simply run this command:

```

$ node test-model.js
..... ✓ OK » 5 honored (1.655s)

```

This is using the dot-matrix test reporter in Vows. If no errors were detected then the output is as shown.

Our code, as it stands, does not have a bug. To show what the output looks like with a failing test case, let's introduce a bug. In `models-sequelize/notes.js`, change the `read` function to this:

```

exports.read = function(key, callback) {
  Note.find({ where: { notekey: key } }).success(function(note) {
    if (!note) {
      callback("Nothing found for " + key);
    } else {
      callback(null, {
        notekey: note.notekey,
        title: "THIS IS A BUG " + note.title,
        body: note.body
      });
    }
  });
}

```

When we rerun the test, the output will be as follows:

```
$ node test-model.js
... ✗ ✗

    read note
      ✗ should have proper note
        »
        actual expected

        THIS IS A BUG Note 1
        // test-model.js:39

    change note after a successful model.update
      ✗ should be changed
        »
        actual expected

        THIS IS A BUG Note 1 title changed
        // test-model.js:53
✗ Broken » 3 honored · 2 broken (2.452s)
```

The output is descriptive enough to tell you which test case failed, what value was expected, what value was received, and how the two differed. There is even some color coding in the terminal window that helps you to see the difference.

Remove the introduced bug, and the test will pass as expected.

Because of the descriptive labels in the test suite, the output is descriptive enough to guide you to the failing test case, and is even somewhat readable.

Testing router functions

Now that we've seen a little bit about testing a data model, let's move on to testing the route functions. There is obviously a lot more to test in the Notes data models, such as the user model. Our goal in this chapter is not to produce a complete test suite, but to introduce you to unit testing in Node.

In unit testing the route functions, we have a little more freedom in mocking out the other modules. We can fairly easily create a faux data model, and can even bypass Express completely. Let's see how it's done.

Let's start with the faux data model. It can also be used as an in-memory data model in the Notes application, if you like. In the `test` directory, create a file named `mock-model.js` containing the following:

```
var events = require('events');
var emitter = module.exports.emitter = new events.EventEmitter();

var notes = [];

exports.connect = function(params, callback) {
  callback();
}

exports.disconnect = function(callback) {
  callback();
}

exports.update = exports.create = function(key, title, body, callback) {
  {
    notes[key] = { title: title, body: body };
    exports.emitter.emit('noteupdated', {
      key: key, title: title, body: body
    });
    callback();
  }
}

exports.read = function(key, callback) {
  callback(undefined, notes[key]);
}

exports.delete = function(key, callback) {
  delete notes[key];
  emitter.emit('notedeleted', key);
  callback();
}

exports.titles = function(callback) {
  var thenotes = [];
  Object.keys(notes).forEach(function (key) {
    thenotes.push({ key: note.notekey, title: note.title });
  });
  callback(null, thenotes);
}
```


This model has exactly the same API as the others, and even generates the events required for the Socket.IO integration. But by storing the note data in memory, we remove one external dependency from the test. In the case of testing route functions, the database is not central to the code being tested, making the argument we made earlier about not mocking out the database irrelevant.

We could, of course, use the test database along with the `Sequelize` model to test the route functions, as we did earlier when testing the `Sequelize` model. But then we would not be testing the route functions in isolation from the model. In theory this is important because an unrelated failure in the database, or in the model code, could surface as a test failure in the router code, making us think there is a router bug when the bug is in a different module.

Now create a file named `test-routes-notes.js` containing the following:

```
var assert = require("assert")
var vows   = require('vows');

var model = require('./mock-model');
var notes = require("../routes/notes");
notes.configure({ model: model });
var setup = require('./setup-model');
setup.config(model, { }, function() { });
```

This part wires the Notes' route functions to use the faux data model we just created. Then we use `setup-model.js` to add the hard-coded test data. This way route functions can be executed, with no dependency on a database or a specific model implementation.

But, how do we execute the route functions? We could fire up a modified `app.js` to get an HTTP server running the Notes application, but that would create unwanted dependencies on the HTTP stack and Express. Instead, what we're going to do is create faux objects to represent the request, response, and next objects, and then just call the route functions directly.

```
var mockRes = function(vows) {
  return {
    render: function() { vows.callback('render', arguments); },
    redirect: function() { vows.callback('redirect', arguments); }
  };
};

var mockNext = function(vows) {
  return function() { vows.callback('next', arguments); };
};
```

These are the faux `response` and `next` objects. They implement the subset of the functionality that our router functions use, the `render` and `redirect` functions. The HTTP Server and Express together provide more functions in the `response` object, so if we were to use those capabilities in our route functions then the `mockRes` object would have to implement those functions as well.

The Vows documentation calls this technique a **macro**, because we can easily use the same code in multiple test contexts. Otherwise we would be duplicating the code in multiple locations, and running the risk of not updating all instances of a given code snippet.

We're using `this.callback` to collect the data that would be sent by the route function to Express. It is called `vows.callback` because the `this` in this place is different from the `this` inside the `topic` of a test context. What we're going to do is pass `this` as the argument called `vows` here, so that calling `vows.callback` is the same as if it were written as `this.callback`.

This trick is a common concern in programming JavaScript. The language nicely builds up a namespace context as you nest code blocks within each other. However, sometimes you have to take extra care like this to ensure the correct object ends up in a useful place where it is needed.

Here are a few tests:

```
vows.describe("notes routes")
  .addBatch({
    "view with bad key": {
      topic: function() {
        notes.view({
          query: { key: "a key" },
          user: "a user"
        }, mockRes(this), mockNext(this));
      },
      "should error on no key": function(command, args) {
        assert.match(command, /render/);
        assert.match(args[0], /showerror/);
        assert.match(args[1].title, /No note found for/);
      }
    },
    "view no key": {
      topic: function() {
        notes.view({
          query: undefined,
          user: "a user"
        }, mockRes(this), mockNext(this));
      },
    },
  });
```

```
    "should error on no key": function(command, args) {
      assert.match(command, /render/);
      assert.match(args[0], /showerror/);
      assert.match(args[1].title, /No key given for/);
    }
  },
  "view correct key": {
    topic: function() {
      notes.view({
        query: { key: "n1" },
        user: "a user"
      }, mockRes(this), mockNext(this));
    },
    "should render no error": function(command, args) {
      assert.match(command, /render/);
      assert.match(args[0], /noteview/);
      assert.equal(args[1].notekey, "n1");
    }
  }
}
}))
.addBatch({
  "save": {
    topic: function() {
      notes.save({
        body: {
          dcreate: "create",
          notekey: "nnew",
          title: "New Note",
          body: "Body of new note"
        },
        user: "a user"
      }, mockRes(this), mockNext(this));
    },
    "should redirect": function(command, args) {
      assert.equal(command, "redirect");
      assert.match(args[0], /noteview/);
    }
  }
}
}).run();
```

In each test context we're calling one of the route functions directly. We're providing an object in place of the request object that Express would have sent, and we can even create arbitrary request objects to exercise whatever corner condition we want to verify. That includes the corner conditions that are unlikely to exist in the wild.

The `mockRes` and `mockNext` functions fill out the rest of the API required to call a route function, and as we just said, help us to capture the data that would be sent by the route functions.

Now, let's run the test suite and see what happens:

```
$ node test-routes-notes.js
XX. .
  view with bad key
    X should error on no key
      » expected null to match /render/ // test-routes-notes.js:33

  view no key
    X should error on no key
      » expected null to match /render/ // test-routes-notes.js:47
X Broken » 2 honored · 2 broken (0.009s)
```

Hmm, we have some errors; what's going on?

Diagnosing a failing test case

We see that the errors are in the `view bad key` and `view no key` test contexts.

The `view bad key` test context uses a `Note` key that we know does not exist in the test data. It's a scenario that, with the current `view` function in `routes/notes.js`, is going to fail. We can see it simply by inspecting the code being tested.

```
exports.view = function(req, res, next) {
  if (req.query.key) {
    readNote(req.query.key, req.user, res, function(err, data) {
      if (!err) {
        res.render('noteview', {
          title: data.title,
          user: req.user ? req.user : undefined,
          notekey: req.query.key,
          note: data
        });
      }
    });
  } else {
    res.render('showerror', {
      title: "No key given for Note",
      user: req.user ? req.user : undefined,
```

```
        error: "Must provide a Key to view a Note"
      });
    }
  }
}
```

This code worked fine with our ad hoc testing in earlier chapters, but with unit testing our tests can be more comprehensive and repeatable. The test case provides a key that does not refer to an existing note. Hence, the `readNote` function is unable to find a note.

After some debugging, we find that the `readNote` function sends `undefined` in both the `err` object and the `data` object. Essentially, the `readNote` function does not indicate an error, and does not send a data object. At the same time, our code assumed it was either going to send an error object, or a data object. A similar mistake was written into the `edit` and `destroy` functions.

One change to do is in the `readNote` function, to make this condition an error:

```
var readNote = function(key, user, res, cb) {
  notes.read(key,
    function(err, data) {
      if (err) {
        res.render('showerror', {
          title: "Could not read note " + key,
          user: user ? user : undefined,
          error: err
        });
        cb(err);
      }
      else if (!data)
        cb(new Error("No note found for " + key));
      else cb(null, data);
    });
}
```

All we've done is insert a clause in the code to raise an error, if no data object is found.

What happens now if we run the test case?

```
$ node test-routes-notes.js
X
X Errored » callback not fired

    in view with bad key
```

```

    in notes routes
    in undefinedX Errored » 1 honored · 1 broken · 1 errored ·
1 dropped

```

It's still raising a problem, but we've made progress because now our program indicates a real error when it's given bad data.

Inspecting the view function again, we see that if the `readNote` function raises an error there is no code to execute. It means the test case never sees a callback to itself that indicates the action taken in this scenario. This is the question we asked early in the chapter; what if the callback we expect to be called for some reason is never called? Additionally, it means that the Notes application itself would behave badly in this case, and not show an error to the user.

The answers a question we had earlier about detection of cases where a callback is never called. Vows manages to detect that `this.callback` was never called, reporting that as an error.

Because there's no code to handle this condition, let's add some suitable code:

```

exports.view = function(req, res, next) {
  var user = req.user ? req.user : undefined;
  if (req.query.key) {
    readNote(req.query.key, user, res, function(err, data){
      if (!err) {
        res.render('noteview', {
          title: data.title,
          user: user,
          notekey: req.query.key,
          note: data
        });
      } else {
        res.render('showerror', {
          title: 'No note found for ' + req.query.key,
          user: user,
          error: "Invalid key " + req.query.key
        });
      }
    });
  } else {
    res.render('showerror', {
      title: "No key given for Note",
      user: user,
      error: "Must provide a Key to view a Note"
    });
  }
}

```

All it takes is an `else` clause and to show an error.

Now let's run the test suite and see what happens:

```
$ node test-routes-notes.js
· X · ·
  view no key
    X should error on no key
      » expected null to match /render/ // test-routes-notes.js:47
X Broken » 3 honored · 1 broken (0.008s)
```

There's still a failing test case, but the `view bad key` test is now succeeding. That's because it successfully detected the error that is now being displayed to the user.

In the `view no key` test case, we aren't providing a query object at all. When our code (again, in the `exports.view` function in `routes/notes.js`) tests whether `req.query.key` exists, an error is triggered because the `key` field cannot be read from the undefined object.

We can simply change that line of code to read as follows:

```
if (req.query && req.query.key) { ...
```

Then our bug is fixed. And now the tests execute correctly:

```
$ node test-routes-notes.js
... · ✓ OK » 4 honored (0.007s)
```

Not only is our test suite now correctly detecting the errors thrown correctly from our program, but we've also found and fixed bugs in the Notes application. Bugs were not found by the sort of ad hoc testing where we declare "it works" before checking in our code. That's what we did in earlier chapters of this book. Run the application through a few ad hoc manual test scenarios, claiming it works, even though it hadn't been fully tested.

Similar bugs exist in the `edit` and `delete` functions, and similar changes should be made in each place. Because our current set of tests are incomplete, the test suite as it stands doesn't catch these bugs. If you want to verify that bugs do exist in the `edit` and `delete` functions, the first step is to write test cases that tickle the bugs. Once you reliably have a failing test case, you can insert the fixes, and the tests should now succeed.

Making it easy to run the tests

Now that we have two test suites for the Notes application, how do we easily run them?

One way is to add a test script to the `package.json` file:

```
"scripts": {
  "start": "node app",
  "test": "node test/test-model.js; node test/test-routes-notes.js"
},
```

On Windows this will be a little bit different:

```
"scripts": {
  "start": "node app",
  "test": "node test/test-model.js & node test/test-routes-notes.js"
},
```

Now we can run both suites together, like so:

```
$ npm test

> application-name@0.0.1 test /Users/david/filez/chap09/notes
> node test/test-model.js; node test/test-routes-notes.js

..... ✓ OK » 5 honored (1.935s)
... . ✓ OK » 4 honored (0.004s)
```

On Windows you might create a batch file, named `test.bat`, containing something like this:

```
node test\test-model.js
node test\test-routes-notes.js
```

Or you could use a Makefile. Put the following into a file named `Makefile`:

```
test: /tmp
  node test/test-model.js
  node test/test-routes-notes.js
```

You, of course, have to make sure the two lines following `test :` start with an actual *Tab* character, because that's the way Makefiles work. Once you have this file, run it like so:

```
$ make test

node test/test-model.js
```



```
..... ✓ OK » 5 honored (1.936s)
node test/test-routes-notes.js
... . ✓ OK » 4 honored (0.004s)
```

Summary

In this concluding chapter, we've covered the basics of developing unit tests for a Node application. Unit testing is of course a large topic, but with this information you can take the steps required to improve your application quality.

Specifically we've covered:

- Considerations for testing asynchronous code (what if the expected callback is never called?)
- Using the assert module to implement quick test cases
- The Vows test framework
- Some questions about when to implement a mock object, and when not to do so
- Creating a test database to hold test data
- Separating the individual units to test in isolation from the rest of the application
- Creating a mock data model, that can also be useful when prototyping the application
- Testing route functions with all dependencies mocked out
- Using test case failures to debug and improve the application
- Making it easy to run the tests

For this to be a complete test suite, it has to do much more than what we've shown here, for example:

- More tests for the Notes module
- Add a test suite for the users data model
- Add a test suite for the other route modules

In addition, developing a complete set of unit tests does not mean your testing is finished. The adage is that testing is never finished, instead it is exhausted. The other major area of testing to consider is functional testing of the complete application.

That is, the easiest method for function testing is to just manually run the application. But this is not repeatable, nor automated, and requires hiring workers to click the mouse around in the application. There are many methods of automating such work, including traditional QA automation tools such as Selenium (<http://docs.seleniumhq.org/>).

An interesting option in the Node community is the zombie module (<https://github.com/assaf/zombie>). Zombie is a mock web browser that can interact with any web application, testing the HTTP responses for given HTTP requests.

In this book, we've covered a lot of ground, showing you the basics of using Node to develop typical web applications, including:

- What is Node, and why it is an important programming platform to learn and use to develop applications
- How to install and use Node
- Developing Node modules
- Using the npm package managers to manage modules
- The event-driven nature of the Node platform
- EventEmitters
- Developing applications at the HTTP interface
- The Express web framework
- The necessity to ensure event handlers execute quickly
- Strategies to refactor your application in case an event handler does runs slowly
- Several varieties of data storage engines
- Developing lightweight REST style services with Node
- User authentication
- Express route middleware to enforce access rights by logged-in users
- Deployment on Linux systems
- Deployment to cloud hosting
- Highly interactive applications where part lives in the browser, part lives in the server, communicating back and forth
- Unit testing

Index

Symbols

`.emit` function 67, 175
`-g` flag 32, 50
`<h1>` tags 108
`.json` extension 120
`.on` function 72, 180
`-prefix` option 27
`.schema` command 127
`<script>` tag 182
`.values` function 63

A

absolute module identifier 40
account page, user authentication
 module, routing for 148-151
add function 118
af command-line tool 168
af tunnel command 169
af update command 167
Alex Russell
 blog, URL 172
algorithm
 refactoring 84-86
app.configure method 78
AppFog
 notes, deploying 163-169
 URL 163
app.get function 78
app.js
 about 31
 changes 142-145
 configuring 122, 130, 134, 135
 configuring, for LevelUP 125
 configuring, for notes 139

app.js module 165
application
 external dependencies, bundling 41-43
 local modules 40, 41
app object 162
app.set method 77
app.use method 77
apt-get update command 20
assert module 201
asynchronous code
 testing 199, 200
asynchronous event-driven architecture
 versus versus threaded architecture 11, 12
async module 86, 122, 123

B

Backbone.js
 URL 172
backend service
 creating 84
bin tag 46

C

callback function 88
Chocolatey Gallery
 URL 23
 used, for installing Node on Windows 23
client code
 setting up 175
client side code
 setting up, for sending messages 190, 191
cloud hosting. *See* AppFog
cluster module 161
Comet application architecture 172

- commands**
 - running 28
 - testing 28
- CommonJS modules**
 - about 62
 - module encapsulation, demonstrating 63
 - URL 62
- configure function** 115, 116, 148
- Connect**
 - about 74
 - URL 74
- connect event** 180
- connect-flash module** 150
- connect function** 120, 132, 175, 188
- connect method** 135
- context** 205
- cookieParser middleware** 143
- cores**
 - using, on multi-core servers 161, 162
- Create App button** 163
- Create button** 164
- create function** 96, 97, 118, 128, 133, 177
- Create operation.** *See* **CRUD model**
- createReadStream function** 125
- CRUD model** 96
- csh users** 26
- curl command** 87, 91

D

- data event** 89
- Data model**
 - used, to store messages 188-190
- data object** 214
- db.all function** 128
- db.each function** 128, 129
- db.get function** 128
- db.run function** 128
- Debian**
 - notes, deploying 158-160
- DELETE command** 129
- deleteFile function** 200
- deleteFile.js file** 200
- delete function** 216
- Delete operation.** *See* **CRUD model**
- delmessage event** 187, 188, 193, 194
- delMessage function** 194

- DerbyJS**
 - URL 172
- destroy function** 96, 119, 134, 153, 178, 214
- Destroy operation.** *See* **CRUD model**
- developer tools**
 - installing from source, for POSIX-like systems 25, 26
 - installing, on Mac OS X 25
- developer tools, installing**
 - multiple Node installs, maintaining 27
- directories tag** 47
- disconnect event** 180, 187
- disconnect function** 120
- doAccount function** 151
- dcreate flag** 101
- dcreate parameter** 118
- dodelmessage event** 188, 193
- dodestroy function** 119
- doLogin function** 149
- doSendMessage function** 191
- duplicate modules**
 - eliminating 52, 53

E

- edit function** 118, 214, 216
- err object** 214
- err variable** 200
- event**
 - EventEmitter class** 65, 66, 180
 - EventEmitter object** 65
 - EventEmitters**
 - about 65
 - events, receiving 65, 66
 - events, sending 65, 66
 - HTTPClient object 65
 - HTTPServer object 65
 - theory 67
- Event Loop** 83
- events**
 - between Notes server and client code 176
 - browser-side event handlers 180-183
 - notdeleted event 176
 - notetitles event 176
 - noteupdated event 176
 - receiving, with EventEmitters 65, 66
 - sending, with EventEmitters 65, 66

- sending, from Notes server 179, 180
- sending, Notes model modified 177, 179
- Everyauth**
 - URL 142
- exports object** 38
- exports variable** 203
- exports.view function** 216
- Express**
 - about 74
 - Fibonacci sequence, calculating 78-82
 - installing 74
 - REST server, implementing 90, 92
 - REST service, calling 89
 - Socket.IO, initializing with 173, 174
 - URL 74
- Express application**
 - about 77, 78
 - look, changing 107-109
- express command** 95, 96
- Express command-line tool** 76
- express module** 42

F

- Favicon** 72
- Fibonacci application**
 - refactoring, for REST 92, 93
- fibonacci function** 86
- Fibonacci sequence**
 - calculating, with Express 78-82
 - code, calculating 83
- fibonum** 82
- fiboserver.js file** 90
- fibotimes.js file** 83
- files**
 - notes, storing 119-121
- findAll function** 189
- findAll operation** 133
- find function** 138
- findOne function** 138
- find operation** 138
- forever tool**
 - about 33, 34, 159
 - URL 33, 159
- fs module** 37
- fs object** 37

G

- getmessages event** 188, 193, 194
- getmessages function** 194
- GET request** 78
- GitHub**
 - URL 163
- globals** 38

H

- handler function** 11
- heartbeat**
 - listening to 185, 186
- help message** 29
- Homebrew**
 - Node, installing on Mac OS X 20
 - URL 20
- homepage tag** 47
- HTTPClient object** 65
- HTTP Client requests**
 - making 87-89
- HTTP conversation**
 - HTTP Sniffer, listening 71-73
- http.createServer function** 68
- http.createServer method** 13
- HTTP request**
 - URL 93
- HTTP request client**
 - URL 93
- http.request method** 88
- HTTP request system**
 - URL 93
- HTTP server application** 67-70
- http.Server object** 68
- HTTPServer object** 65
- HTTP Sniffer**
 - listening, to HTTP conversation 71-73

I

- images directory** 107
- index.ejs template** 115
- index.ejs view** 114
- index function** 151
- init script** 35, 160
- INSERT INTO command** 97

instances
multiple, running 109-111
multiple, scaling up 109-111

Intel's paper
URL 17

J

JavaScript
about 9, 10, 171
advantages 10
disadvantages 10
javascripts directory 107
JOIN query 145

K

key parameter 122

L

launchd script 34
LevelDB
URL 123
LevelUP
about 123
app.js, configuring 125
installing 123
model code, for notes 124, 125
URL 123
used, for storing notes 123
lib directory 47
LinkedIn
URL 14
Linux
node, installing from package management systems 21
listen function 31, 68, 72
listen method 13
local modules 40, 41
Local strategy 142
location parameter 124
login, user authentication
module, routing for 148-151
logout, user authentication
module, routing for 148-151
LSB-style init script
URL 159

M

Mac OS X
developer tools, installing 25
Node, installing with Homebrew 20
Node, installing with MacPorts 20
MacPorts
Node, installing on Mac OS X 20
project, URL 20
macro 211
main tag 46
Makefile file 217
messageBox element 194
messages
dispatching, between client-side 192, 193
dispatching, between server-side 192, 193
displaying, to user 193-195
sending 196
sending, between users 187
sending, client side code setting up 190, 191
sent between users, Socket.IO events used 187
storing, data model used 188-190
method field 88
middleware 73
Mikito Takada blog
URL 14
mock-model.js file 209
mockNext function 213
mockRes function 213
mockRes object 211
model
testing 202-207
model code 131-134
model configuration
injecting, into routers 116
models directory 96
model-view-controller. *See* MVC
module encapsulation
demonstrating 63
module identifier
about 39
absolute module identifier 40
relative module identifier 40
top-level module identifier 40
modules
about 37, 38

- as directories 44, 45
- complex modules 44
- identifiers 39
- path names 39
- routing, for account page 148, 149
- routing, for login 148-150
- routing, for logout 148, 149
- MongoDB**
 - about 135
 - notes, storing with Mongoose 135
 - URL 135
- mongohosting**
 - URL 135
- mongohq**
 - URL 135
- mongolab**
 - URL 135
- Mongoose**
 - about 136
 - notes model, implementing 136-138
 - notes, storing in MongoDB 136
 - URL 136
- multi-core servers**
 - cores, using 161, 162
- multiple Node installation**
 - maintaining 27
- MVC**
 - about 73, 95, 96
 - routes directory 95
 - views directory 95
- MySQL button 164**

N

- newmessage event 187, 189, 193, 194**
- next function 38**
- next object 211**
- Node**
 - about 7, 8
 - algorithm, for resolving require('module') 39
 - algorithmic refactoring 84
 - architecture 11, 12
 - backend service, creating 84
 - capabilities 8
 - command-line tools 28, 29
 - distribution, installing from nodejs.org 22, 23
 - green web hosting 16
 - installing from source, on POSIX-like systems from source 24
 - installing, URL 24
 - modules 37, 39
 - performance 13-15
 - performance case study, URL 14
 - prerequisites, installing 24
 - server, launching 30, 31
 - server-side JavaScript 9
 - server, starting at system start-up 33-36
 - server utilization 16
 - simple script, running 29, 30
 - system requisites 19
 - URL 17, 30
 - using 10
 - utilization 13, 14
- node command 24, 28, 202**
- NODE_ENV variable 78**
- Nodefront**
 - URL 27
- node-gyp project**
 - installing, URL 51, 124
- node-init**
 - about 33
 - URL 33
- Node, installing**
 - on Linux, from package management systems 21
 - on Mac OS X, with MacPorts 20
 - on Mac OS X, with with Homebrew 20
 - on Windows, Chocolatey Gallery used 23
 - package managers used 20
- Node.js. See Node**
- Node.JS. See Node**
- nodejs.org**
 - Node distribution, installing 22, 23
- node_modules**
 - duplicate modules, eliminating 52, 53
- Node modules 93**
- node_modules directory 40, 42, 43, 50**
- Node package manager. See npm**
- NODE_PATH variable**
 - about 43
 - system-wide modules 43, 44

- notetitles event function** 181
- Node version manager**
 - URL 27
- node-waf tool** 28
- notdeleted event** 176, 178, 180, 183
- noteedit.ejs template** 104
- notekey object** 100
- notekey parameter** 183
- note object** 100, 103, 133
- Notes**
 - adding 99, 101
 - deleting 105, 106
 - deploying, on AppFog 163-169
 - deploying, on Debian 158-160
 - editing 104
 - home page 97-99
 - model 96, 97
 - real-time web features, adding 172
 - running 196
 - viewing 102, 103
- Notes application**
 - asynchronizing 114, 115
 - code, creating 96
 - running 155-158
 - running, with Socket.IO 183
- Notes application, asynchronizing**
 - model configuration, injecting into routers 116
 - notes router 116-119
- Notes application code, creating**
 - Notes, adding 99-101
 - Notes, deleting 105, 107
 - Notes, editing 104
 - Notes home page 97, 98
 - Notes model 96, 97
 - Notes, viewing 102, 103
- notes.destroy** 119
- Notes model**
 - implementing, in Mongoose 136-138
 - modifying, to send events 177, 179
- notes module** 98
- notes router** 116-119
- Notes server**
 - events, sending from 179, 180
- notes, storing**
 - app.js, configuring 122, 123, 129, 130, 134, 135, 139
 - app.js, configuring for LevelUP 125, 126
 - in files 119-122
 - in MongoDB with Mongoose 135, 136
 - in SQL 126
 - LevelUP, installing 123
 - LevelUP model code, for notes 124, 125
 - model code 127-129
 - notes model implementation, in Mongoose 136, 138
 - schema, setting up with SQLite3 126, 127
 - schema, setup 131-134
 - with LevelUP data store 123
 - with Sequelize module 131
- Notes.sync() function** 132
- notes.titles function** 115
- notetitles event** 176
- noteupdated event** 176-178, 180, 183
- npm**
 - about 31, 32, 45
 - CommonJS modules 62
 - configuration settings 59
 - URL 49
- npm command**
 - about 24, 49
 - duplicate modules, eliminating 52, 53
 - installed npm package, uninstalling 57
 - installed packages, listing 53, 54
 - native code modules, installing on Windows 51
 - outdated packages, updating 56
 - package information, viewing 49
 - packages, installing to module 51
 - using 48
- npm config set command** 60
- npm get command** 59
- npm init command** 57
- npm install command** 50, 58
- npm list command** 53
- npm package**
 - about 45
 - developing 57, 58
 - finding 47
 - format 45, 46
 - information, viewing 49
 - installed content, editing 55, 56
 - installed content, exploring 55, 56
 - installed packages, listing 53, 54

- installing 50
- installing, to module 51
- outdated packages, updating 56
- publishing 57, 58
- ranges 60, 61
- scripts 55
- uninstalling 57
- version strings 60, 61

npm publish command 59

npm repository
URL 46

npm set command 59

npm unpublish command 59

npm view command 47

O

Object-Relation Mapping. *See* **ORM**

openid module 45

OpenSSL
URL 19

options object 88

ORM 131

P

package.json file 44, 45, 161, 217

package managers
URL 21

used, for installing Node 20

params.model 116

Passport
URL 142

passport-local 142

Passport middleware 143

passport module 148

path.join function 121

path module 121

path names 39

PATH variable 26

Percolator.JS
URL 93

persistent storage
Create operation 96
Read operation 96
Update operation 96

POSIX-like systems
Node, installing from source 24-26

postLogin function 151

POST operation 101

public directory
about 107
images directory 107
javascripts directory 107
stylesheets directory 107

Pulser class 66

pulse event 66

pulser.js file 66

Pulser object 66

Python
URL 19

Q

qs module 42

Quality Engineering (QE) 199

query function 12

R

readdirSync function 30

read function 96, 122, 137, 138, 207

readNote function 152, 214, 215

Read operation. *See* **CRUD model**

real-time web features
adding, to Notes 172

redirect function 211

relative module identifier 40

remove function 138

removeListener function 180

render function 211

Representational State Transfer (REST) 87

req.flash function 150

req.isAuthenticated function 149

request event 68

request.method field 70

request object 68, 70, 212

request.url field 70

require function 37, 62, 63

require statement 41, 97, 116

req.user variable 154

response object 68, 89, 211

res.render function 82

REST client library
URL 93

- REST server**
 - Fibonacci application, refactoring 92, 93
 - implementing, with Express 90, 92
- REST service**
 - calling, from Express application 89
- route middleware 145**
- router functions**
 - testing 208-212
- router module 148**
- routers**
 - about 73
 - model configuration, injecting 116
- routes directory 95**
- routes/notes.js module 151**
- Ruby**
 - installer, URL 167
- Ryan Dahl**
 - URL 13

S

- save function 101, 118**
- schema**
 - setting up, with sqlite3 126
 - setup 131-134
- script**
 - running, with Node 29, 30
- script tag 47**
- sequelize-based users model 146, 147**
- Sequelize model 210**
- Sequelize module**
 - notes, storing with 131
 - URL 131
- sequelize-params.js file 203**
- sequelize-params module 144**
- server**
 - launching, with Node 30, 31
- session middleware 142, 143**
- setInterval 66**
- setup function 205**
- setup.js tool 168**
- ShareJS**
 - URL 172
- showerror.ejs template 115**
- simple.js module 38**
- slnode command 24**
- Small-scale web applications 14**

- sniffOn function 72**
- Socket.IO**
 - about 173
 - client code, setting up 175
 - initializing, with Express 173, 174
 - Notes application, running with 183-185
- Socket.IO events**
 - used, for sending messages between users 187
- socket object 175**
- special versions 61**
- SQLite3**
 - about 126
 - advantages 126
 - app.js, configuring 129, 130
 - model code 127-129
 - schema, setting up 126
 - URL 126
- sqlite3 command 126**
- sqlite3 module 127**
- sqlite3 package 126**
- start method 66**
- start script 55**
- state 141**
- static file servers 73**
- strategies 142**
- StrongLoop Node**
 - about 23
 - installing 23
 - URL 23
- stylesheets directory 107**
- success function 133**

T

- test.bat file 217**
- test case**
 - diagnosing 213-216
- test-deleteFile.js file 200, 201**
- test driven development (TDD) 202**
- test-model.js file 203**
- tests**
 - executing 207, 208
 - running 217
- test/setup-model.js script 206**
- this.callback function 206**

- threaded architecture**
 - versus asynchronous event-driven architecture 11, 12
- title parameter** 122
- titles function** 122, 125, 134, 138
- topic field** 205
- top-level module identifier** 40
- tunnel** 168
- Twitter Bootstrap**
 - URL 108

U

- Ubuntu Upstart tool**
 - about 34
 - URL 34
- undefined object** 216
- unique: false attribute** 188
- unit testing** 199
- updateAttributes function** 133
- UPDATE command** 97
- update function** 96, 97, 118, 128, 138, 178
- updatenote function** 183
- Update operation.** *See* **CRUD model**
- USAGE message** 29
- user authentication**
 - about 141, 142
 - app.js, changes 142-145
 - module, routing for account page 148-151
 - module, routing for login 148-151
 - module, routing for logout 148-151
 - Notes application, running 155-158
 - sequelize-based users model 146, 147
 - user table, initializing 154
- user.js module** 80
- user object** 151-154
- users.ensureAuthenticated function** 145, 156
- users module** 145
- user table**
 - initializing 154
- user variable** 153

V

- V8** 161
- VCAP_APP_PORT environment variable** 165
- VCAP_SERVICES environment variable** 164, 166
- view function** 213
- views directory** 95, 100, 103, 151
- views/fibonacci.ejs file** 80
- vows.describe function** 205
- Vows framework**
 - URL 202

W

- Web frameworks**
 - about 73, 74
 - need for 74
 - URL 73
- wget command** 87
- Windows**
 - native code modules, installing 51
 - Node installing, Chocolatey Gallery used 23
- workforce.js module** 165
- workforce module** 161
- WritableStream**
 - URL 89

X

- Xcode**
 - URL 25

Y

- Yahoo Axis**
 - URL 14
- YSlow**
 - URL 83



Thank you for buying **Node Web Development *Second Edition***

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.

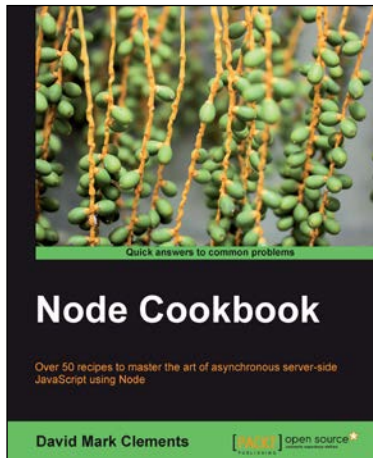
About Packt Open Source

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.



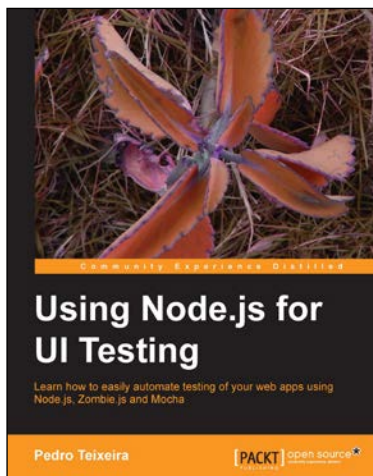
Node Cookbook

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



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

Please check www.PacktPub.com for information on our titles

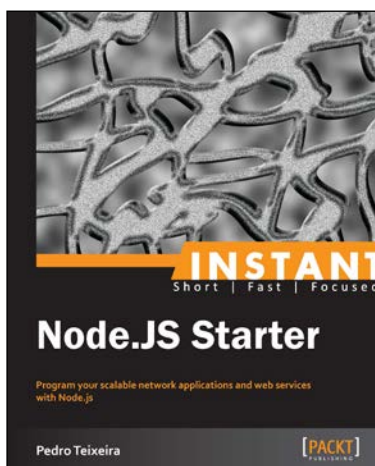


Socket.IO Real-time Web Application Development

ISBN: 978-1-78216-078-6 Paperback: 140 pages

Build modern real-time web applications powered by Socket IO

1. Understand the usage of various socket.io features like rooms, namespaces, and sessions
2. Secure the socket.io communication
3. Deploy and scale your socket.io and Node.js applications in production
4. A practical guide that quickly gets you up and running with socket.io



Instant Node.js Starter

ISBN: 978-1-78216-556-9 Paperback: 48 pages

Program your scalable network applications and web services with Node.js

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results
2. Learn how to use module patterns and Node Package Manager (NPM) in your applications
3. Discover callback patterns in NodeJS
4. Understand the use Node.js streams in your applications

Please check www.PacktPub.com for information on our titles