



Quick answers to common problems

Node Cookbook

Over 50 recipes to master the art of asynchronous server-side
JavaScript using Node

David Mark Clements

www.it-ebooks.info

PACKT open source 
PUBLISHING community experience distilled

Node Cookbook

Over 50 recipes to master the art of asynchronous
server-side JavaScript using Node

David Mark Clements



BIRMINGHAM - MUMBAI

Node Cookbook

Copyright © 2012 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: July 2012

Production Reference: 1120712

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

ISBN 978-1-84951-718-8

www.packtpub.com

Cover Image by Asher Wishkerman (a.wishkerman@mpic.de)

Credits

Author

David Mark Clements

Project Coordinator

Sai Gamare

Reviewers

Marc Harter

Joachim Larsen

Joshua Holbrook

Proofreader

Joel T. Johnson

Indexer

Rekha Nair

Acquisition Editor

Sarah Cullington

Graphics

Manu Joseph

Lead Technical Editor

Jonathan Titmus

Production Coordinator

Shantanu Zagade

Technical Editor

Manasi Poonthottam

Cover Work

Shantanu Zagade

About the Author

David Mark Clements is a web entrepreneur residing in Northern Ireland. From a very early age he was fascinated with programming and computers. He first learned BASIC on one of the many Atari's he had accumulated by the age of nine. David learned JavaScript at the age of 12, moving into Linux administration and PHP as a teenager.

Now (as a twenty-something) he uses his technical skills in business, providing a variety of services ranging from tailored websites, to complete online strategies, to implementing custom warehouse logistics integrated with e-commerce solutions. Node has become a prominent member of his toolkit due to its versatility and powerful interfacing abilities.

When he's not tinkering with computers, he's working in a homeless hostel, playing bass with Living Rivers worship team, or spending time with the love of his life, Maxine, and her Husky-Spitz cross, Jessi.

Many thanks to the Node community, who have caused Node to grow as it has, and Node Google Group, which has been an immense source of information and inspiration.

I cannot conclude without acknowledging Jesus, who makes my life worthwhile and gave me strength to complete this book among all the other commitments, interruptions, and general life events.

About the Reviewers

Marc Harter is a Senior Web Developer for Applied Data Consultants in Eau Claire, WI. He is actively involved in the JavaScript community from the browser to the server side. He enjoys studying programming languages and paradigms, obsessively changing his theme and background on ArchLinux, and keeping up with the latest in web development. He has a beautiful wife and two beautiful children.

Joachim Larsen is interested in making stuff work. His appetite for learning new skills and applying them in new ways has seen him build total-solution systems for computer cafes; Win95 Active Desktop HTML frontend, Samba backend with save-game propagation, Windows Registry manipulation—including CD key pools for optimizing the use of purchased game inventory, networked-based CD image mounting, and more. Being a restless sort, he took to working offshore, building and installing boxes for getting the most out of satellite-based Internet connections while ensuring a hassle-free experience for the individual users.

Currently, his focus is on making the user experience faster, and making it faster to make more user experiences—generally, in the context of JavaScript-rich web applications.

He co-hosts the <http://javascriptjabber.com/> podcast and dabbles in applying electronics to small-scale food production.

I would like to thank my parents for a very interesting childhood that continues to inspire and inform. I love you guys :)

Joshua Holbrook, who hails from the far North, earned a master's degree in Mechanical Engineering at the University of Alaska Fairbanks, where he successfully defended his thesis on *The measurement of anisotropic thermal conductivity in snow with needle probes*. His original background is in MATLAB and Python, but he has since mastered Node.js. He has now moved to sunny California with his bird Korben to work in startups.

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://www.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: Making a Web Server	5
Introduction	5
Setting up a router	5
Serving static files	10
Caching content in memory for immediate delivery	15
Optimizing performance with streaming	19
Securing against filesystem hacking exploits	24
Chapter 2: Exploring the HTTP Object	29
Introduction	29
Processing POST data	29
Handling file uploads	34
Using Node as an HTTP client	40
Implementing download throttling	45
Chapter 3: Working with Data Serialization	51
Introduction	51
Converting an object to JSON and back again	51
Converting an object to XML and back again	56
Browser-server transmission via AJAX	65
Working with real data: fetching trending tweets	74
Chapter 4: Interfacing with Databases	81
Introduction	81
Writing to a CSV file	82
Connecting and sending SQL to a MySQL server	85
Storing and retrieving data with MongoDB	90
Storing and retrieving data with Mongoskin	96
Storing data to CouchDB with Cradle	99

Table of Contents

Retrieving data from CouchDB with Cradle	102
Accessing CouchDB changes stream with Cradle	108
Storing and retrieving data with Redis	111
Implementing PubSub with Redis	115
Chapter 5: Transcending AJAX: Using WebSockets	121
Introduction	121
Creating a WebSocket server	122
Seamless fallbacking with socket.io	128
Callbacks over socket.io transport	134
Creating a real-time widget	138
Chapter 6: Accelerating Development with Express	147
Introduction	147
Generating Express scaffolding	148
Defining and applying environments	152
Dynamic routing	156
Templating in Express	160
CSS engines with Express	169
Initializing and using a session	178
Making an Express web app	187
Chapter 7: Implementing Security, Encryption, and Authentication	209
Introduction	209
Implementing Basic Authentication	210
Cryptographic password hashing	213
Implementing Digest Authentication	218
Setting up an HTTPS web server	224
Preventing cross-site request forgery	227
Chapter 8: Integrating Network Paradigms	237
Introduction	237
Sending email	238
Sending SMS	242
Communicating with TCP	249
Creating an SMTP server	256
Implementing a virtual hosting paradigm	261

Table of Contents

Chapter 9: Writing Your Own Node Modules	269
Introduction	269
Creating a test-driven module API	270
Writing a functional module mock-up	275
Refactoring from functional to prototypical	279
Extending the module's API	286
Deploying a module to npm	296
Chapter 10: Taking It Live	301
Introduction	301
Deploying to a server environment	301
Automatic crash recovery	307
Continuous deployment	310
Hosting with a Platform as a Service provider	318
Index	323

Preface

The principles of asynchronous event-driven programming are perfect for today's Web, where efficient real-time applications and scalability are at the forefront. Server-side JavaScript has been here since the nineties, but Node got it right. With a thriving community and interest from the Internet giants, it could be the PHP of tomorrow.

Node Cookbook shows you how to transfer your JavaScript skills to server-side programming. With simple examples and supporting code, *Node Cookbook* takes you through various server-side scenarios, often saving you time, effort, and trouble by demonstrating best practices and showing you how to avoid security mistakes.

Beginning with making your own web server, the practical recipes in this cookbook are designed to smoothly progress you to making full web applications, command-line applications, and Node modules. *Node Cookbook* takes you through interfacing with various database backends, such as MySQL, MongoDB, and Redis, working with web sockets, and interfacing with network protocols, such as SMTP. Additionally, there are recipes on handling streams of data, security implementations, writing your own Node modules, and different ways to take your apps live.

What this book covers

Chapter 1, Making a Web Server, covers serving dynamic and static content, caching files in memory, streaming large files straight from disk over HTTP, and securing your web server.

Chapter 2, Exploring the HTTP Object, explains receiving and processing POST requests and file uploads, using Node as an HTTP client, and discusses how to throttle downloads.

Chapter 3, Working with Data Serialization, explains how to convert data from your apps into XML and JSON formats for sending to the browser or third-party APIs.

Chapter 4, Interfacing with Databases, covers how to implement persistent data stores with Redis, CouchDB, MongoDB, MySQL, or plain CSV files.

Chapter 5, Transcending AJAX: Using WebSockets, helps you to make real-time web apps with modern browser WebSocket technology, gracefully degrade to long-polling and other methods with `Socket.io`.

Chapter 6, Accelerating Development with Express, explains how to leverage the Express framework to achieve rapid web development. It also covers using template languages and CSS engines, such as LESS and Stylus.

Chapter 7, Implementing Security, Encryption, and Authentication, explains how to set up an SSL secured web server, use the crypto module to create strong password hashes, and protect your users from cross-site request forgery attacks.

Chapter 8, Integrating Network Paradigms, discusses sending emails and creating your own email server, sending SMS text messages, implementing virtual hosting, and doing fun and interesting things with raw TCP.

Chapter 9, Writing Your Own Node Modules explains how to create a test suite, write a solution, refactor, improve and extend, and then deploy your own Node module.

Chapter 10, Taking It Live discusses how to deploy your web apps to a live server, ensure your apps stay live with crash recovery techniques, implement a continuous deployment work flow, or alternatively simply use a Platform as a Service Provider.

What you need for this book

- ▶ Windows, Mac OS X, or Linux
- ▶ Node 0.6.x or Node 0.8.x freely obtained from <http://www.nodejs.org>

Will continue to be relevant on Node's 1.x.x release

Who this book is for

If you have some knowledge of JavaScript and want to build fast, efficient, scalable client-server solutions, then *Node Cookbook* is for you. Experienced users of Node will improve their skills, and even if you have not worked with Node before, these practical recipes will make it easy to get started.

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: "In order to create the server we need the `http` module."

A block of code is set as follows:

```
var http = require('http');
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/html'});
  response.end('Woohoo!');
}).listen(8080);
```

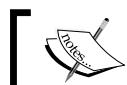
When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
var http = require('http');
var path = require('path');
http.createServer(function (request, response) {
  var lookup=path.basename(decodeURI(request.url));
```

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

```
sudo npm -g install express
```

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: "We could make it possible for a hypothetical user to indicate if they were inspired by a quote, such as a **Like** 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 book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com/suggest@packtpub.com

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/support>, 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

Making a Web Server

In this chapter we will cover:

- ▶ Setting up a router
- ▶ Serving static files
- ▶ Caching content in memory for immediate delivery
- ▶ Optimizing performance with streaming
- ▶ Securing against filesystem hacking exploits

Introduction

One of the great qualities of Node is its simplicity. Unlike PHP or ASP there is no separation between the web server and code, nor do we have to customize large configuration files to get the behavior we want. With Node we can create the server, customize it, and deliver content all at the code level. This chapter demonstrates how to create a web server with Node and feed content through it, all while implementing security and performance enhancements to cater for various situations.

Setting up a router

In order to deliver web content we need to make a URI available. This recipe walks us through the creation of an HTTP server that exposes routes to the user.

Getting ready

First, let's create our server file. If our main purpose is to expose server functionality, it's general practice to call the file `server.js`, which we could put in a new folder. It's also a good idea to install and use hot node:

```
sudo npm -g install hot node
hot node server.js
```

Hot node will conveniently auto-restart the server when we save changes.

How to do it...

In order to create the server we need the `http` module, so let's load it and use the `http.createServer` method:

```
var http = require('http');
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/html'});
  response.end('Woohoo!');
}).listen(8080);
```

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 emailed directly to you.

Now, if we save our file and access `localhost:8080` on a web browser or using curl, our browser (or curl) will exclaim: 'Woohoo!'. However, the same will occur at `localhost:8080/foo`. Indeed, any path will render the same behavior, so let's build in some routing. We can use the `path` module to extract `basename` of the path (the final part of the path), and reverse any URI encoding from the client with `decodeURI`:

```
var http = require('http');
var path = require('path');
http.createServer(function (request, response) {
  var lookup = path.basename(decodeURI(request.url));
```

We now need a way to define our routes. One option is to use an array of objects:

```
var pages = [
  {route: '', output: 'Woohoo!'},
  {route: 'about', output: 'A simple routing with Node example'},
  {route: 'another page', output: function() {return 'Here\'s ' + this.route;}},
];
```

Our `pages` array should be placed above the `http.createServer` call.

Within our server, we need to loop through our array and see if the `lookup` variable matches any of our routes. If it does we can supply the output. We'll also implement some 404 handling:

```
http.createServer(function (request, response) {
  var lookup = path.basename(decodeURI(request.url));
  pages.forEach(function(page) {
    if (page.route === lookup) {
      response.writeHead(200, {'Content-Type': 'text/html'});
      response.end(typeof page.output === 'function'
        ? page.output() : page.output);
    }
  });
  if (!response.finished) {
    response.writeHead(404);
    response.end('Page Not Found!');
  }
}).listen(8080);
```

How it works...

The callback function we provide to `http.createServer` gives us all the functionality we need to interact with our server through the `request` and `response` objects. We use `request` to obtain the requested URL and then we acquire its `basename` with `path`. We also use `decodeURI` which our `another page` route would fail without as our code would try to match `another %20page` against our `pages` array and return `false`.

Once we have our `basename`, we can match it in any way we want. We could send it in a database query to retrieve content, use regular expressions to effectuate partial matches, or we could match it to a file name and load its contents.

We could have used a `switch` statement to handle routing but our `pages` array has several advantages. It's easier to read and extend, and it can be seamlessly converted to JSON. We loop through our `pages` array using `for Each`.

Node is built on Google's V8 engine, which provides us with a number of ECMAScript 5 features. These features can't be used in all browsers as they're not yet universally implemented, but using them in Node is no problem! `for Each` is an ES5 implementation, but the ES3 way is to use the less convenient `for` loop.

While looping through each object, we check its `route` property. If we get a match, we write the 200 `OK` status and `content-type` headers. We then end the response with the object's output property.

`response.end` allows us to pass a parameter to it, which it writes just before finishing the response. In `response.end`, we used a ternary operator (`? :`) to conditionally call `page.out put` as a function or simply pass it as a string. Notice that the `another page` route contains a function instead of a string. The function has access to its parent object through the `this` variable, and allows for greater flexibility in assembling the output we want to provide. In the event that there is no match in our `for Each` loop, `response.end` would never be called. Therefore, the client would continue to wait for a response until it times out. To avoid this, we check the `response.finished` property and if it's false, we write a 404 header and end the response.

`response.finished` depends on the `for Each` callback, yet it's not nested within the callback. Callback functions are mostly used for asynchronous operations. So on the surface this looks like a potential race condition, however `for Each` does not operate asynchronously. It continues to block until all loops are complete.

There's more...

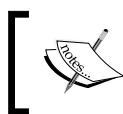
There are many ways to extend and alter this example. There's also some great non-core modules available that do the leg work for us.

Simple multilevel routing

So far, our routing only deals with a single-level path. A multilevel path (for example, `/about/node`) will simply return a 404. We can alter our object to reflect a subdirectory-like structure, remove `path`, and use `request.url` for our routes instead of `path.basename`:

```
var http=require('http');
var pages = [
  {route: '/', out put: 'Woohoo!'},
  {route: '/about/this', out put: 'Multilevel routing with Node'},
  {route: '/about/node', out put: 'Evented I/O for V8 JavaScript.'},
  {route: '/another page', out put: function () {return 'Here\'s ' + this.route;}}
```

```
];
http.createServer(function (request, response) {
  var lookup = decodeURI(request.url);
```



When serving static files, `request.url` must be cleaned prior to fetching a given file. Check out the *Securing against filesystem hacking exploits* section discussed in this chapter.



Multilevel routing could be taken further, allowing us to build and then traverse a more complex object.

```
{route: 'about', childRoutes: [
  {route: 'node', output: 'Evented I/O for V8 Javascript'},
  {route: 'this', output: 'Complex Multilevel Example'}
]}
```

After the third or fourth level, this object would become a leviathan to look at. We could instead create a helper function to define our routes that essentially pieces our object together for us. Alternatively, we could use one of the excellent non-core routing modules provided by the open source Node community. Excellent solutions already exist which provide helper methods to handle the increasing complexity of scalable multilevel routing (see *Routing modules* discussed in this chapter and *Chapter 6, Accelerating Development with Express*).

Parsing the querystring

Two other useful core modules are `url` and `querystring`. The `url.parse` method allows two parameters. First the URL string (in our case, this will be `request.url`) and second a Boolean parameter named `parseQueryString`. If set to `true`, it lazy loads the `querystring` module, saving us the need to require it, to parse the query into an object. This makes it easy for us to interact with the query portion of a URL.

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

var pages = [
  {id: '1', route: '', output: 'Woohoo!'},
  {id: '2', route: 'about', output: 'A simple routing with Node example'},
  {id: '3', route: 'another page', output: function () {return
    'Here\'s ' + this.route; }},
];

http.createServer(function (request, response) {
  var id = url.parse(decodeURI(request.url), true).query.id;
```

```
if (id) {
  pages.forEach(function (page) {
    if (page.id === id) {
      response.writeHead(200, {'Content-Type': 'text/html'});
      response.end(typeof page.output === 'function'
        ? page.output() : page.output);
    }
  });
}

if (!response.finished) {
  response.writeHead(404);
  response.end('Page Not Found');
}

}).listen(8080);
```

With the added `id` properties we can access our object data by, for instance, `localhost:8080?id=2`.

Routing modules

There's an up-to-date list of various routing modules for Node at <https://www.github.com/joeyent/node/wiki/modules#wiki-web-frameworks-routers>. These community-made routers cater to various scenarios. It's important to research the activity and maturity of a module before taking it into a production environment. In *Chapter 6, Accelerating Development with Express*, we will go into greater detail on using the built-in Express/Connect router for more comprehensive routing solutions.

See also

- ▶ *Serving static files* and *Securing against filesystem hacking exploits* discussed in this chapter
- ▶ *Dynamic Routing* discussed in *Chapter 6, Accelerating Development with Express*

Serving static files

If we have information stored on disk that we want to serve as web content, we can use the `fs` (filesystem) module to load our content and pass it through the `createServer` callback. This is a basic conceptual starting point for serving static files. As we will learn in the following recipes there are much more efficient solutions.

Getting ready

We'll need some files to serve. Let's create a directory named `content`, containing the following three files:

`index.html`:

```
<html>
<head>
<title>Yay Node!</title>
<link rel="stylesheet" href="styles.css" type="text/css">
<script src="script.js" type="text/javascript"></script>
</head>
<body>
<span id="yay">Yay!</span>
</body>
</html>
```

`script.js`:

```
window.onload=function() {alert('Yay Node!');};
```

`styles.css`:

```
#yay {font-size: 5em; background: blue; color: yellow; padding: 0.5em}
```

How to do it...

As in the previous recipe, we'll be using the core modules `http` and `path`. We'll also need to access the filesystem, so we'll require the `fs` module too. Let's create our server:

```
var http = require('http');
var path = require('path');
var fs = require('fs');
http.createServer(function (request, response) {
  var lookup = path.basename(decodeURI(request.url)) || 'index.html',
    f = 'content/' + lookup;
  fs.exists(f, function (exists) {
    console.log(exists ? lookup + " is there" : lookup + " doesn't exist");
  });
}).listen(8080);
```

If we haven't already, we can initialize our `server.js` file:

```
hot node server.js
```

Making a Web Server

Try loading `localhost:8080/foo` and the console will say `foo doesn't exist`, because it doesn't. `localhost:8080/script.js` will tell us `script.js is there`, because it is. Before we can save a file, we are supposed to let the client know the `content-type`, which we can determine from the file extensions. So let's make a quick map using an object:

```
var mimeTypes = {  
  '.js' : 'text/javascript',  
  '.html': 'text/html',  
  '.css' : 'text/css'  
};
```

We could extend our `mimeTypes` map later to support more types.

 Modern browsers may be able to interpret certain mime types (such as `text/javascript`) without the server sending a `content-type` header. However, older browsers or less common mime types will rely upon the correct `content-type` header being sent from the server.

Remember to place `mimeTypes` outside the server callback since we don't want to initialize the same object on every client request. If the requested file exists, we can convert our file extension into `content-type` by feeding `path.extname` into `mimeTypes` and then passing our retrieved `content-type` to `response.writeHead`. If the requested file doesn't exist, we'll write out a 404 and end the response.

```
// requires variables, mimeType object...  
http.createServer(function (request, response) {  
  var lookup = path.basename(decodeURI(request.url)) || 'index.html',  
    f = 'content/' + lookup;  
  fs.exists(f, function (exists) {  
    if (exists) {  
      fs.readFile(f, function (err, data) {  
        if (err) { response.writeHead(500);  
          response.end('Server Error!!'); return; }  
        var headers = {'Content-type': mimeTypes[path.  
          extname(lookup)]};  
        response.writeHead(200, headers);  
        response.end(data);  
      });  
    }  
  return;
```

```

        }
        response.writeHead(404); // no such file found!
        response.end();
    });
}).listen(8080);

```

At the moment, there is still no content sent to the client. We have to get this content from our file, so we wrap the response handling in an `fs.readFile` method callback.

```

// http.createServer, inside fs.exists:
if (exists) {
    fs.readFile(f, function(err, data) {
        var headers={'Content-type': mimeTypes[path.extname(lookup)]};
        response.writeHead(200, headers);
        response.end(data);
    });
    return;
}

```

Before we finish, let's apply some error handling to our `fs.readFile` callback as follows:

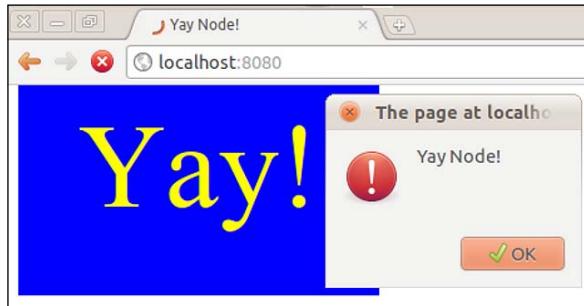
```

// requires variables, mimeType object...
// http.createServer, fs.exists, inside if(exists):
fs.readFile(f, function(err, data) {
    if (err) {response.writeHead(500); response.end('Server Error!');}
    return;
    var headers = {'Content-type': mimeTypes[path.extname(lookup)]};
    response.writeHead(200, headers);
    response.end(data);
});
return;
}

```

Notice that `return` stays outside the `fs.readFile` callback. We are returning from the `fs.exists` callback to prevent further code execution (for example, sending 404). Placing `return` in an `if` statement is similar to using an `else` branch. However, the `if return` pattern is generally preferable to using `if else` in Node, as it eliminates yet another set of curly braces.

So now we can navigate to `localhost:8080` which will serve our `index.html` file. The `index.html` file makes calls to our `script.js` and `styles.css` files, which our server also delivers with appropriate mime types. The result can be seen in the following screenshot:



This recipe serves to illustrate the fundamentals of serving static files. Remember, this is not an efficient solution! In a real-world situation, we don't want to make an I/O call every time a request hits the server, this is very costly especially with larger files. In the following recipes, we'll learn better ways to serve static files.

How it works...

Our script creates a server and declares a variable called `lookup`. We assign a value to `lookup` using the double pipe (`||`) operator. This defines a default route if `path.basename` is empty. Then we pass `lookup` to a new variable that we named `f` in order to prepend our `content` directory to the intended filename. Next we run `f` through the `fs.exists` method and check the `exist` parameter in our callback to see if the file is there. If the file exists we read it asynchronously using `fs.readFile`. If there is a problem accessing the file, we write a 500 server error, end the response, and return from the `fs.readFile` callback. We can test the error-handling functionality by removing read permissions from `index.html`.

```
chmod -r index.html
```

Doing so will cause the server to throw the 500 server error status code. To set things right again run the following command:

```
chmod +r index.html
```

As long as we can access the file, we grab `content-type` using our handy `mimeTypes` mapping object, write the headers, end the response with data loaded from the file, and finally return from the function. If the requested file does not exist, we bypass all this logic, write a 404, and end the response.

There's more...

Here's something to watch out for...

The favicon gotcha

When using a browser to test our server, sometimes an unexpected server hit can be observed. This is the browser requesting the default `favicon.ico` icon file that servers can provide. Apart from the initial confusion of seeing additional hits, this is usually not a problem. If the favicon request begins to interfere, we can handle it like this:

```
if (request.url === '/favicon.ico') {  
    response.end();  
    return;  
}
```

If we wanted to be more polite to the client, we could also inform it of a 404 by using `response.writeHead(404)` before issuing `response.end()`.

See also

- ▶ *Caching content in memory for immediate delivery* discussed in this chapter
- ▶ *Optimizing performance with streaming* discussed in this chapter
- ▶ *Securing against filesystem hacking exploits* discussed in this chapter

Caching content in memory for immediate delivery

Directly accessing storage on each client request is not ideal. For this example, we will explore how to enhance server efficiency by accessing the disk on only the first request, caching the data from file for that first request, and serving all further requests out of the process memory.

Getting ready

We are going to improve upon the code from the previous task, so we'll be working with `server.js`, and in the `content` directory with `index.html`, `styles.css`, and `script.js`.

How to do it...

Let's begin by looking at our script from the previous recipe *Serving static files*:

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

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

var mimeTypes = {
  '.js' : 'text/javascript',
  '.html': 'text/html',
  '.css' : 'text/css'
};

http.createServer(function (request, response) {
  var lookup = path.basename(decodeURI(request.url)) || 'index.html';
  var f = 'content/' + lookup;
  fs.exists(f, function (exists) {
    if (exists) {
      fs.readFile(f, function(err, data) {
        if (err) { response.writeHead(500);
                    response.end('Server Error!'); return; }
        var headers = {'Content-type': mimeTypes[path.extname(lookup)]};
        response.writeHead(200, headers);
        response.end(data);
      });
      return;
    }
    response.writeHead(404); // no such file found!
    response.end('Page Not Found!');
  });
});
```

We need to modify this code to only read the file once, load its contents into memory, and afterwards respond to all requests for that file from memory. To keep things simple and preserve maintainability, we'll extract our cache handling and content delivery into a separate function. So above `http.createServer`, and below `mimeTypes`, we'll add the following:

```
var cache = {};
function cacheAndDeliver(f, cb) {
  if (!cache[f]) {
    fs.readFile(f, function(err, data) {
      if (!err) {
        cache[f] = {content: data};
      }
      cb(err, data);
    });
    return;
  }
  console.log('loading ' + f + ' from cache');
  cb(null, cache[f].content);
}
// http.createServer ....
```

A new `cache` object has been added to store our files in memory as well as a new function called `cacheAndDeliver`. Our function takes the same parameters as `fs.readFile`, so we can replace `fs.readFile` in the `http.createServer` callback while leaving the rest of the code intact:

```
//...inside http.createServer:  
fs.exists(f, function(exists) {  
    if (exists) {  
        cacheAndDeliver(f, function(err, data) {  
            if (err) { response.writeHead(500);  
                response.end('Server Error!'); return; }  
            var headers = {'Content-type': mimeTypes[path.extname(f)]};  
            response.writeHead(200, headers);  
            response.end(data);  
        });  
        return;  
    }  
    //rest of fs.exists code (404 handling)...  
});
```

When we execute our `server.js` file and access `localhost:8080` twice consecutively, the second request causes the console to output the following:

```
Loading content/index.html from cache  
Loading content/styles.css from cache  
Loading content/script.js from cache
```

How it works...

We defined a function called `cacheAndDeliver`, which like `fs.readFile`, takes a filename and callback as parameters. This is great because we can pass the exact same callback of `fs.readFile` to `cacheAndDeliver`, padding the server out with caching logic without adding any extra complexity visually to the inside of the `http.createServer` callback. As it stands, the worth of abstracting our caching logic into an external function is arguable, but the more we build on the server's caching abilities the more feasible and useful this abstraction becomes. Our `cacheAndDeliver` function checks to see if the requested content is already cached, if not, we call `fs.readFile` and load the data from disk. Once we have this data we may as well hold onto it, so it's placed into the `cache` object referenced by its file path (the `f` variable). The next time anyone requests the file, `cacheAndDeliver` will see that we have the file stored in the `cache` object and will issue an alternative callback containing the cached data. Notice that we fill the `cache[f]` property with another new object containing a `content` property. This makes it easier to extend the caching functionality in the future since we would just need to place extra properties into our `cache[f]` object and supply logic that interfaces with these properties accordingly.

There's more...

If we were to modify the files we are serving, any changes wouldn't be reflected until we restart the server. We can do something about that.

Reflecting content changes

To detect whether a requested file has changed since we last cached it, we must know when the file was cached and when it was last modified. To record when the file was last cached, let's extend the `cache[f]` object:

```
cache[f] = {content: data,
            timestamp: Date.now() // store a Unix time stamp
          };
```

That was easy. Now we need to find out when the file was updated last. The `fs.stat` method returns an object as the second parameter of its callback. This object contains the same useful information as the command-line GNU coreutils `stat`. `fs.stat` supplies three time-related properties: `last accessed` (`atime`), `last modified` (`mtime`), and `last changed` (`ctime`). The difference between `mtime` and `ctime` is that `ctime` will reflect any alterations to the file, whereas `mtime` will only reflect alterations to the content of the file. Consequently, if we changed the permissions of a file, `ctime` would update but `mtime` would stay the same. We want to pay attention to permission changes as they happen, so let's use the `ctime` property:

```
// requires and mimeType object...
var cache = {};
function cacheAndDeliver(f, cb) {
  fs.stat(f, function (err, stats) {
    var lastChanged = Date.parse(stats.ctime),
        isUpdated = (cache[f]) && lastChanged > cache[f].timestamp;
    if (!cache[f] || isUpdated) {
      fs.readFile(f, function (err, data) {
        console.log('loading ' + f + ' from file');
        // rest of cacheAndDeliver
      });
    }
  });
}
```

The contents of `cacheAndDeliver` have been wrapped in an `fs.stat` callback. Two variables have been added and the `if(!cache[f])` statement has been modified. We parse the `ctime` property of the second parameter, dub `stats` using `Date.parse` to convert it to milliseconds since midnight, January 1, 1970 (the Unix epoch), and assign it to our `lastChanged` variable. Then we check whether the requested file's last changed time is greater than when we cached the file (provided the file is indeed cached) and assign the result to our `isUpdated` variable. After that, it's merely a case of adding the `isUpdated` Boolean to the conditional `if(!cache[f])` statement via the `||` (or) operator. If the file is newer than our cached version (or if it isn't yet cached), we load the file from the disk into the `cache` object.

See also

- ▶ *Optimizing performance with streaming* discussed in this chapter
- ▶ *Browser-server transmission via AJAX* discussed in Chapter 3, *Working with Data Serialization*

Optimizing performance with streaming

Caching content certainly improves upon reading a file from disk for every request. However, with `fs.readFile`, we are reading the whole file into memory before sending it out in `response`. For better performance, we can stream a file from disk and pipe it directly to the `response` object, sending data straight to the network socket one piece at a time.

Getting ready

We are building on our code from the last example, so let's get `server.js`, `index.html`, `styles.css`, and `script.js` ready.

How to do it...

We will be using `fs.createReadStream` to initialize a stream, which can be piped to the `response` object. In this case, implementing `fs.createReadStream` within our `cacheAndDeliver` function isn't ideal because the event listeners of `fs.createReadStream` will need to interface with the `request` and `response` objects. For the sake of simplicity, these would preferably be dealt within the `http.createServer` callback. For brevity's sake, we will discard our `cacheAndDeliver` function and implement basic caching within the server callback:

```
// requires, mime types, createServer, lookup and vars...
fs.exists(f, function (exists) {
  if (exists) {
    var headers = {'Content-type': mimeTypes[path.extname(f)]};
    if (cache[f]) {
      response.writeHead(200, headers);
      response.end(cache[f].content);
      return;
    } //...rest of server code...
```

Later on, we will fill `cache[f].content` while we're interfacing with the `readStream` object. Here's how we use `fs.createReadStream`

```
var s = fs.createReadStream(f);
```

Making a Web Server

This will return a `readStream` object which streams the file that is pointed at by the `f` variable. `readStream` emits events that we need to listen to. We can listen with `addEventListener` or use the shorthand on:

```
var s = fs.createReadStream(f).on('open', function () {
  // do stuff when the readStream opens
});
```

Since `createReadStream` returns the `readStream` object, we can latch our event listener straight onto it using method chaining with the dot notation. Each stream is only going to open once, we don't need to keep on listening to it. Therefore, we can use the `once` method instead of `on` to automatically stop listening after the first event occurrence:

```
var s = fs.createReadStream(f).once('open', function () {
  // do stuff when the readStream opens
});
```

Before we fill out the `open` event callback, let's implement error handling as follows:

```
var s = fs.createReadStream(f).once('open', function () {
  // do stuff when the readStream opens
}).once('error', function (e) {
  console.log(e);
  response.writeHead(500);
  response.end('Server Error!');
});
```

The key to this entire endeavor is the `stream.pipe` method. This is what enables us to take our file straight from disk and stream it directly to the network socket via our `response` object.

```
var s = fs.createReadStream(f).once('open', function () {
  response.writeHead(200, headers);
  this.pipe(response);
}).once('error', function (e) {
  console.log(e);
  response.writeHead(500);
  response.end('Server Error!');
});
```

What about ending the response? Conveniently, `stream.pipe` detects when the stream has ended and calls `response.end` for us. For caching purposes, there's one other event we need to listen to. Still within our `fs.exists` callback, underneath the `createReadStream` code block, we write the following code:

```
fs.stat(f, function(err, stats) {
  var bufferOffset = 0;
```

```

cache[f] = {content: new Buffer(stats.size)};
s.on('data', function(chunk) {
  chunk.copy(cache[f].content, bufferOffset);
  bufferOffset += chunk.length;
});
);

```

We've used the `data` event to capture the buffer as it's being streamed, and copied it into a buffer that we supplied to `cache[f].content`, using `fs.stat` to obtain the file size for the file's cache buffer.

How it works...

Instead of the client waiting for the server to load the entire file from the disk prior to sending it to the client, we use a stream to load the file in small, ordered pieces and promptly send them to the client. With larger files this is especially useful, as there is minimal delay between the file being requested and the client starting to receive the file.

We did this by using `fs.createReadStream` to start streaming our file from the disk. `fs.createReadStream` creates `readStream`, which inherits from the `EventEmitter` class.

The `EventEmitter` class accomplishes the *evented* part of Node's tag line: Evented I/O for V8 JavaScript. Due to this, we'll use listeners instead of callbacks to control the flow of stream logic.

Then we added an `open` event listener using the `once` method since we want to stop listening for `open` once it has been triggered. We respond to the `open` event by writing the headers and using the `stream.pipe` method to shuffle the incoming data straight to the client.

`stream.pipe` handles the data flow. If the client becomes overwhelmed with processing, it sends a signal to the server which should be honored by pausing the stream. Under the hood, `stream.pipe` uses `stream.pause` and `stream.resume` to manage this interplay.

While the response is being piped to the client, the content cache is simultaneously being filled. To achieve this, we had to create an instance of the `Buffer` class for our `cache[f].content` property. A `Buffer` must be supplied with a size (or an array or string) which in our case is the size of the file. To get the size, we used the asynchronous `fs.stat` and captured the `size` property in the callback. The `data` event returns `Buffer` as its only callback parameter.

The default `bufferSize` for a stream is 64 KB. Any file whose size is less than the `bufferSize` will only trigger one `data` event because the entire file will fit into the first chunk of data. However, for files greater than `bufferSize`, we have to fill our `cache[f].content` property one piece at a time.

Changing the default `readStream` buffer size:

We can change the buffer size of `readStream` by passing an `options` object with a `bufferSize` property as the second parameter of `fs.createReadStream`.

For instance, to double the buffer you could use `fs.createReadStream(f, {bufferSize: 128 * 1024});`



We cannot simply concatenate each chunk with `cache[f].content` since this will coerce binary data into string format which, though no longer in binary format, will later be interpreted as binary. Instead, we have to copy all the little binary buffer chunks into our binary `cache[f].content` buffer.

We created a `bufferOffset` variable to assist us with this. Each time we add another chunk to our `cache[f].content` buffer, we update our new `bufferOffset` by adding the length of the chunk buffer to it. When we call the `Buffer.copy` method on the chunk buffer, we pass `bufferOffset` as the second parameter so our `cache[f].content` buffer is filled correctly.

Moreover, operating with the `Buffer` class renders performance enhancements with larger files because it bypasses the V8 garbage collection methods. These tend to fragment large amounts of data thus slowing down Node's ability to process them.

There's more...

While streaming has solved a problem of waiting for files to load into memory before delivering them, we are nevertheless still loading files into memory via our `cache` object. With larger files, or large amounts of files, this could have potential ramifications.

Protecting against process memory overruns

There is a limited amount of process memory. By default, V8's memory is set to 1400 MB on 64-bit systems and 700 MB on 32-bit systems. This can be altered by running Node with `--max-old-space-size=N` where N is the amount of megabytes (the actual maximum amount that it can be set to depends upon the OS and of course the amount of physical RAM available). If we absolutely needed to be memory intensive, we could run our server on a large cloud platform, divide up the logic, and start new instances of node using the `child_process` class.

In this case, high memory usage isn't necessarily required and we can optimize our code to significantly reduce the potential for memory overruns. There is less benefit to caching larger files. The slight speed improvement relative to the total download time is negligible while the cost of caching them is quite significant in ratio to our available process memory. We can also improve cache efficiency by implementing an expiration time on cache objects which can then be used to clean the cache, consequently removing files in low demand and prioritizing high-demand files for faster delivery. Let's rearrange our `cache` object slightly:

```
var cache = {
  store: {},
  maxSize : 26214400, // (bytes) 25mb
}
```

For a clearer mental model, we're making a distinction between the cache as a functioning entity and the cache as a store (which is a part of the broader cache entity). Our first goal is to only cache files under a certain size. We've defined `cache.maxSize` for this purpose. All we have to do now is insert an `if` condition within the `fs.stat` callback:

```
fs.stat(f, function (err, stats) {
  if (stats.size < cache.maxSize) {
    var bufferOffset = 0;
    cache.store[f] = {content: new Buffer(stats.size),
                      timestamp: Date.now() };
    s.on('data', function (data) {
      data.copy(cache.store[f].content, bufferOffset);
      bufferOffset += data.length;
    });
  }
});
```

Notice we also slipped in a new `timestamp` property into our `cache.store[f]`. This is for cleaning the cache, which is our second goal. Let's extend `cache`:

```
var cache = {
  store: {},
  maxSize: 26214400, // (bytes) 25mb
  maxAge: 5400 * 1000, // (ms) 1 and a half hours
  clean: function(now) {
    var that = this;
    Object.keys(this.store).forEach(function(file) {
      if (now > that.store[file].timestamp + that.maxAge) {
        delete that.store[file];
      }
    });
  }
};
```

So in addition to `maxSize`, we've created a `maxAge` property and added a `clean` method. We call `cache.clean` at the bottom of the server like so:

```
// all of our code prior
cache.clean(Date.now());
}).listen(8080); // end of the http.createServer
```

`cache.clean` loops through `cache.store` and checks to see if it has exceeded its specified lifetime. If it has, we remove it from `store`. We'll add one further improvement and then we're done. `cache.clean` is called on each request. This means `cache.store` is going to be looped through on every server hit, which is neither necessary nor efficient. It would be better if we cleaned the cache, say, every two hours or so. We'll add two more properties to `cache`. The first is `cleanAfter` to specify how long between cache cleans. The second is `cleanedAt` to determine how long it has been since the cache was last cleaned.

```
var cache = {
  store: {},
  maxSize: 26214400, // (bytes) 25mb
  maxAge: 5400 * 1000, // (ms) 1 and a half hours
  cleanAfter: 7200 * 1000, // (ms) two hours
  cleanedAt: 0, // to be set dynamically
  clean: function (now) {
    if (now - this.cleanedAt > this.cleanedAt) {
      this.cleanedAt = now;
      that = this;
      Object.keys(this.store).forEach(function (file) {
        if (now > that.store[file].timestamp + that.maxAge) {
          delete that.store[file];
        }
      });
    }
  }
};
```

We wrap our `cache.clean` method in an `if` statement which will allow a loop through `cache.store` only if it has been longer than two hours (or whatever `cleanAfter` is set to), since the last clean.

See also

- ▶ *Handling file uploads* discussed in Chapter 2, *Exploring the HTTP Object*.
- ▶ *Securing Against Filesystem Hacking Exploits* discussed in this chapter.

Securing against filesystem hacking exploits

For a Node app to be insecure, there must be something an attacker can interact with for exploitation purposes. Due to Node's minimalist approach, the onus is mostly on programmers to ensure their implementation doesn't expose security flaws. This recipe will help identify some security risk anti-patterns that could occur when working with the filesystem.

Getting ready

We'll be working with the `same content` directory as in the previous recipes, but we'll start a new `insecure_server.js` file (there's a clue in the name!) from scratch to demonstrate mistaken techniques.

How to do it...

Our previous static file recipes tend to use `path.basename` to acquire a route, but this flat levels all request. If we accessed `localhost:8080/foo/bar/styles.css`, our code would take `styles.css` as the `basename` and deliver `content/styles.css` to us. Let's make a subdirectory in our `content` folder, call it `subcontent`, and move our `script.js` and `styles.css` files into it. We'd need to alter our script and link tags in `index.html`:

```
<link rel="stylesheet" type="text/css" href="subcontent/styles.css">
<script src="subcontent/script.js" type="text/javascript"></script>
```

We can use the `url` module to grab the entire pathname. So let's include the `url` module in our new `insecure_server.js` file, create our HTTP server, and use `pathname` to get the whole requested path:

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

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

http.createServer(function (request, response) {
  var lookup = url.parse(decodeURI(request.url)).pathname;
  lookup = (lookup === "/") ? '/index.html' : lookup;
  var f = 'content' + lookup;
  console.log(f);
  fs.readFile(f, function (err, data) {
    response.end(data);
  });
}).listen(8080);
```

If we navigate to `localhost:8080`, everything works great. We've gone multilevel, hooray. For demonstration purposes, a few things have been stripped out from previous recipes (such as `fs.exists`), but even with them, the following code presents the same security hazards:

```
curl localhost:8080/.../insecure_server.js
```

Now we have our server's code. An attacker could also access /etc/passwd with a few attempts at guessing its relative path:

```
curl localhost:8080/../../../../../../../../etc/passwd
```

In order to test these attacks, we have to use curl or another equivalent because modern browsers will filter these sorts of requests. As a solution, what if we added a unique suffix to each file we wanted to serve and made it mandatory for the suffix to exist before the server coughs it up? That way, an attacker could request /etc/passwd or our `insecure_server.js` because they wouldn't have the unique suffix. To try this, let's copy the `content` folder and call it `content-pseudosafe`, and rename our files to `index.html-serve`, `script.js-serve`, and `styles.css-serve`. Let's create a new server file and name it `pseudosafe_server.js`. Now all we have to do is make the `-serve` suffix mandatory:

```
// requires section...
http.createServer(function (request, response) {
  var lookup = url.parse(decodeURI(request.url)).pathname;
  lookup = (lookup === "/") ? '/index.html-serve' : lookup + '-serve';
  var f = 'content-pseudosafe' + lookup;
```

For feedback purposes, we'll also include some 404 handling with the help of `fs.exists`.

```
// requires, create server etc
fs.exists(f, function (exists) {
  if (!exists) {
    response.writeHead(404);
    response.end('Page Not Found!');
    return;
  }
// read file etc
```

So let's start our `pseudosafe_server.js` file and try out the same exploit:

```
curl -i localhost:8080/../../insecure_server.js
```

We've used the `-i` argument so that curl will output the headers. What's the result? A 404, because the file it is actually looking for is `..../insecure_server.js-serve`, which doesn't exist. So what's wrong with this method? Well it's inconvenient and prone to error. However, more importantly an attacker can still work around it!

```
curl localhost:8080/../../insecure_server.js%00/index.html
```

And voila! There's our server code again. The solution to our problem is `path.normalize`, which cleans up our `pathname` before it gets to `fs.readFile`.

```
http.createServer(function (request, response) {
  var lookup = url.parse(decodeURI(request.url)).pathname;
  lookup = path.normalize(lookup);
  lookup = (lookup === "/") ? '/index.html' : lookup;
  var f = 'content' + lookup
```

Prior recipes haven't used `path.normalize`, yet they're still relatively safe. `path.basename` gives us the last part of the path, so any leading relative directory pointers (`.. /`) are discarded, thus preventing the directory traversal exploit.

How it works...

Here we have two filesystem exploitation techniques: the **relative directory traversal** and **poison null byte attacks**. These attacks can take different forms, such as in a POST request or from an external file. They can have different effects. For example, if we were writing to files instead of reading them, an attacker could potentially start making changes to our server. The key to security in all cases is to validate and clean any data that comes from the user. In `insecure_server.js`, we pass whatever the user requests to our `fs.readFile` method. This is foolish because it allows an attacker to take advantage of the relative path functionality in our operating system by using `.. /`, thus gaining access to areas that should be off limits. By adding the `-serve` suffix, we didn't solve the problem. We put a plaster on it which can be circumvented by the poison null byte. The key to this attack is `%00`, which is a URL hex code for the null byte. In this case, the null byte blinds Node to the `.. /insecure_server.js` portion, but when the same null byte is sent through to our `fs.readFile` method, it has to interface with the kernel. However, the kernel gets blinded to the `index.html` part. So our code sees `index.html` but the read operation sees `.. /insecure_server.js`. This is known as null byte poisoning. To protect ourselves, we could use a `regex` statement to remove the `.. /` parts of the path. We could also check for the null byte and spit out a `400 Bad Request` statement. However, we don't need to, because `path.normalize` filters out the null byte and relative parts for us.

There's more...

Let's further delve into how we can protect our servers when it comes to serving static files.

Whitelisting

If security was an extreme priority, we could adopt a strict whitelisting approach. In this approach, we would create a manual route for each file we are willing to deliver. Anything not on our whitelist would return 404. We can place a `whitelist` array above `http.createServer` as shown in the following code:

```
var whitelist = [
  '/index.html',
  '/subcontent/styles.css',
  '/subcontent/script.js'
];
```

Inside of our `http.createServer` callback, we'll put an `if` statement to check if the requested path is in the `whitelist` array:

```
if (whitelist.indexOf(lookup) === -1) {  
  response.writeHead(404);  
  response.end('Page Not Found!');  
  return;  
}
```

That's it. We can test this by placing a file `non-whitelisted.html` in our `content` directory.

```
curl -i localhost:8080/non-whitelisted.html
```

The preceding command will return 404 because `non-whitelisted.html` isn't on whitelist.

Node-static

`https://github.com/joeyent/node/wiki/modules#wiki-web-frameworks-static` has a list of static file server modules available for different purposes. It's a good idea to ensure that a project is mature and active before relying on it to serve your content. Node-static is a well developed module with built-in caching. It's also compliant with the RFC2616 HTTP standards specification. This defines how files should be delivered over HTTP. Node-static implements all the essentials discussed in this chapter and more besides. This piece of code is slightly adapted from the `node-static` Github page at `https://github.com/cloudhead/node-static`:

```
var static = require('node-static');  
var fileServer = new static.Server('./content');  
require('http').createServer(function (request, response) {  
  request.addListener('end', function () {  
    fileServer.serve(request, response);  
  });  
}).listen(8080);
```

The preceding code will interface with the `node-static` module to handle server-side and client-side caching, use streams to deliver content, and filter out relative requests and null bytes, among other things.

See also

- ▶ Preventing cross-site request forgery discussed in *Chapter 7, Implementing Security, Encryption, and Authentication*
- ▶ Setting up an HTTPS web server discussed in *Chapter 7, Implementing Security, Encryption, and Authentication*
- ▶ Deploying to a server environment discussed in *Chapter 10, Taking It Live*
- ▶ Cryptographic password hashing discussed in *Chapter 7, Implementing Security, Encryption, and Authentication*

2

Exploring the HTTP Object

In this chapter we will cover:

- ▶ Processing POST data
- ▶ Handling file uploads
- ▶ Using Node as an HTTP client
- ▶ Implementing download throttling

Introduction

In the previous chapter, we used the `http` module to create a web server. Now we're going to look into some associated use cases beyond simply pushing content from server to client. The first three recipes will explore how to receive data via client-initiated HTTP POST (and PUT) requests, and in the final recipe we'll demonstrate how to throttle a stream of outbound data.

Processing POST data

If we want to be able to receive POST data, we have to instruct our server on how to accept and handle a POST request. In PHP we could access our POST values seamlessly with `$_POST['field name']`, because it would block until an array value was filled. By contrast, Node provides low-level interaction with the flow of HTTP data allowing us to interface with the incoming message body as a stream, leaving it entirely up to the developer to turn that stream into usable data.

Getting ready

Let's create a `server.js` file ready for our code, and an HTML file called `form.html`, containing the following code:

```
<form method="post">
  <input type="text" name="userinput1"><br>
  <input type="text" name="userinput2"><br>
  <input type="submit">
</form>
```



For our purposes, we'll place `form.html` in the same folder as `server.js`, though this is not generally a recommended practice. Usually, we should place our public code in a separate folder from our server code.

How to do it...

We'll provision our server for both GET and POST requests. Let's start with GET by requiring the `http` module and loading `form.html` for serving through `createServer`:

```
var http = require('http');
var form = require('fs').readFileSync('form.html');
http.createServer(function (request, response) {
  if (request.method === "GET") {
    response.writeHead(200, {'Content-Type': 'text/html'});
    response.end(form);
  }
}).listen(8080);
```

We are synchronously loading `form.html` at initialization time instead of accessing the disk on each request. If we navigate to `localhost:8080`, we'll be presented with a form. However, if we fill out our form nothing happens because we need to handle POST requests:

```
if (request.method === "POST") {
  var postData = '';
  request.on('data', function (chunk) {
    postData += chunk;
  }).on('end', function() {
    console.log('User Posted:\n' + postData);
    response.end('You Posted:\n' + postData);
  });
}
```

Once the form is completed and submitted, the browser and console will output the raw query string sent from the client. Converting `postData` into an object provides an easy way to interact with and manipulate the submitted information. The `querystring` module has a `parse` method which transforms query strings into objects, and since form submission arrives in query string format, we can use it to objectify our data as follows:

```
var http = require('http');
var querystring = require('querystring');
var util = require('util');
var form = require('fs').readFileSync('form.html');

http.createServer(function (request, response) {
  if (request.method === "POST") {
    var postData = '';
    request.on('data', function (chunk) {
      postData += chunk;
    }).on('end', function () {
      var postDataObject = querystring.parse(postData);
      console.log('User Posted:\n', postData);
      response.end('You Posted:\n' + util.inspect(postDataObject));
    });
  }
  if (request.method === "GET") {
    response.writeHead(200, {'Content-Type': 'text/html'});
    response.end(form);
  }
}).listen(8080);
```

Notice the `util` module. We require it to use its `inspect` method for a simple way to output our `postDataObject` to the browser.

Finally, we're going to protect our server from memory overload exploits.

Protecting a POST server

 V8 (and therefore Node) has virtual memory limitations, based upon the processor architecture and operating system constraints. These limitations far exceed the demands of most use cases. Nevertheless, if we don't restrict the amount of data our POST server will accept, we could leave ourselves open for a type of Denial of Service attack. Without protection, an extremely large POST request could cause our server to slow down significantly or even crash.

To achieve this, we'll set a variable for the maximum acceptable data size and check it against the growing length of our post Data variable.

```
var http = require('http');
var querystring = require('querystring');
var util = require('util');
var form = require('fs').readFileSync('form.html');
var maxData = 2 * 1024 * 1024; // 2mb
http.createServer(function (request, response) {
  if (request.method === "POST") {
    var postData = '';
    request.on('data', function (chunk) {
      postData += chunk;
      if (postData.length > maxData) {
        postData = '';
        this.pause();
        response.writeHead(413); // Request Entity Too Large
        response.end('Too large');
      }
    }).on('end', function () {
      if (!postData) { response.end(); return; } // prevents empty post
      requests from crashing the server
      var postDataObject = querystring.parse(postData);
      console.log('User Posted:\n', postData);
      response.end('You Posted:\n' + util.inspect(postDataObject));
    });
  }
});
```

// rest of our code....

How it works...

Once we know a POST request has been made of our server (by checking `request.method`), we aggregate our incoming data into our `postData` variable via the `data` event listener on the `request` object. However, if we find that the submitted data exceeds our `maxData` limit, we will clear our `postData` variable and pause the incoming stream preventing any further data arriving from the client. Using `stream.destroy` instead of `stream.pause` seems to interfere with our response mechanism. Once a stream has been paused for a while it is automatically removed from memory by v8's garbage collector.

Then we send a 413 Request Entity Too Large HTTP header. In the `end` event listener, as long as `postData` hasn't been cleared for exceeding `maxData` (or wasn't blank in the first place), we use `querystring.parse` to turn our POST message body into an object. From this point, we could perform any number of interesting activities: manipulate, analyze, pass it to a database, and so on. However, for the example, we simply output `postDataObject` to the browser and `postData` to the console.

There's more...

If we want our code to look a little more elegant, and we're not so concerned about handling POST data as a stream, we can employ a user land (non-core) module to get a little sugar on our syntax.

Accessing POST data with `connect bodyParser`

Connect is an excellent middleware framework for Node providing a method framework that assimilates a higher level of abstraction for common server tasks. Connect is actually the basis of the Express web framework, which will be discussed in *Chapter 6, Accelerating Development with Express*.

One piece of middleware that comes bundled with Connect is `bodyParser`. By chaining `connect(bodyParser)` to a normal callback function, we suddenly have access to the POST data via `request.body` (when data is sent by the POST request it is held in the message body). `request.body` turns out to be exactly the same object as `postDataObject` we generated in our recipe.

First, let's make sure we have Connect installed:

```
npm install connect
```

We require `connect` in place of `http` since it provides us with the `createServer` capabilities. To access the `createServer` method, we can use `connect.createServer`, or the shorthand version, which is simply `connect`. Connect allows us to combine multiple pieces of middleware together by passing them in as parameters to the `createServer` method. Here's how to implement similar behavior, as in the recipe using Connect:

```
var connect = require('connect');
var util = require('util');
var form = require('fs').readFileSync('form.html');
connect(connect.limit('64kb'), connect.bodyParser(),
  function (request, response) {
    if (request.method === "POST") {
      console.log('User Posted:\n', request.body);
      response.end('You Posted:\n' + util.inspect(request.body));
    }
    if (request.method === "GET") {
      response.writeHead(200, {'Content-Type': 'text/html'});
      response.end(form);
    }
  }).listen(8080);
```

Notice we are no longer using the `http` module directly. We pass `connect.limit` in as our first parameter to achieve the same `maxData` restriction implemented in the main example.

Next, we pass in `bodyParser`, allowing `connect` to retrieve our POST data for us, objectifying the data into `request.body`. Finally, there's our callback function, with all the former POST functionality stripped out except the code to echo our data object (which is now `request.body`) to console and browser. This is where we deviate slightly from our original recipe.

In the recipe we return the raw `postData` to the console, though we return the `request.body` object here. To output raw data with Connect would either take pointless deconstruction of our object to reassemble the raw query string or an extension of the `bodyParser` function. This is the tradeoff with using third-party modules: we can only easily interact with information the module author expects us to interact with.

Let's look under the hood for a moment. If we fire up an instance of `node` without any arguments, we can access the REPL (Read-Eval-Print-Loop) which is the Node command-line environment. In the REPL, we can write:

```
console.log(require('connect').bodyParser.toString());
```

If we look at the output, we'll see its `connect.bodyParser` function code and should be able to easily identify the essential elements from our recipe at work in the `connect.bodyParser` code.

See also

- ▶ *Handling file uploads* discussed in this chapter
- ▶ *Browser-server transmission via AJAX* discussed in Chapter 3, *Working with Data Serialization*
- ▶ *Initializing and using a session* discussed in Chapter 6, *Accelerating Development with Express*

Handling file uploads

We cannot process an uploaded file in the same way we process other POST data. When a file input is submitted in a form, the browser processes the file into a **multipart message**.

Multipart was originally developed as an email format allowing multiple pieces of mixed content to be combined into one message. If we intuitively attempted to receive the upload as a stream and write it to a file, we would have a file filled with multipart data instead of the file or files themselves. We need a multipart parser, the writing of which is more than a recipe can cover. So instead we'll be using the well-known and battle-tested `formidable` module to convert our upload data into files.

Getting ready

Let's create a new `uploads` directory for storing uploaded files and get ready to make modifications to our `server.js` file from the previous recipe.

We'll also need to install `formidable` as follows:

```
npm install formidable@1.x.x
```

Finally, we'll make some changes to our `form.html` from the last recipe:

```
<form method=POST enctype=multipart/form-data>
<input type=file name=userfile1><br>
<input type=file name=userfile2><br>
<input type=submit>
</form>
```

We've included an `enctype` attribute of `multipart/form-data` to signify to the browser that the form will contain upload data and we've replaced the text inputs with file inputs.

How to do it...

Let's see what happens when we use our modified form to upload a file to the server from the last recipe. Let's upload `form.html` itself as our file:

```
Node Server
File Edit View Search Terminal Help
hotnode: server.js changed
hotnode: Node.js process restarted
hotnode: server.js changed
hotnode: Node.js process restarted
User Posted:
-----WebKitFormBoundaryw9lhsJW9CK5SaaRA
Content-Disposition: form-data; name="userfile1"; filename="form.html"
Content-Type: text/html

<form method=POST enctype=multipart/form-data>
<input type=file name=userfile1><br>
<input type=file name=userfile2><br>
<input type=submit>
</form>

-----WebKitFormBoundaryw9lhsJW9CK5SaaRA
Content-Disposition: form-data; name="userfile2"; filename=""
Content-Type: application/octet-stream

-----WebKitFormBoundaryw9lhsJW9CK5SaaRA--
```

Our POST server simply logs the raw HTTP message body to the console, which in this case is multipart data. We had two file inputs on the form. Though we only uploaded one file, the second input is still included in the multipart request. Each file is separated by a predefined boundary that is set in a secondary attribute of the Content - Type HTTP headers. We'll need to use `formidable` to parse this data, extracting each file contained therein.

```
var http = require('http');
var formidable = require('formidable');
var form = require('fs').readFileSync('form.html');

http.createServer(function (request, response) {
  if (request.method === "POST") {
    var incoming = new formidable.IncomingForm();
    incoming.uploadDir = 'uploads';
    incoming.on('file', function (field, file) {
      if (!file.size) { return; }
      response.write(file.name + ' received\n');
    }).on('end', function () {
      response.end('All files received');
    });
    incoming.parse(request);
  }
  if (request.method === "GET") {
    response.writeHead(200, {'Content-Type': 'text/html'});
    response.end(form);
  }
}).listen(8080);
```

Our POST server has now become an upload server.

How it works...

We create a new instance of the `formidable IncomingForm` class and tell it where to upload files. In order to provide feedback to the user, we can listen to our `incoming` instance. The `IncomingForm` class emits its own higher level events, so rather than listening to the `request` object for events and processing data as it comes, we wait for `formidable` to parse the files out of the multipart message and then notify us through its custom `file` event.

The `file` event callback provides us with two parameters: `field` and `file`. The `file` parameter is an object containing information about the uploaded file. We use this to filter out empty files (usually caused by empty input fields) and grab the filename which we show to users as confirmation. When `formidable` has finished parsing the multipart message, it sends an `end` event in which we end the response.

There's more...

We can post more than simple form fields and values from a browser. Let's take a look at transferring files from browser to server.

Using formidable to accept all POST data

`formidable` doesn't just handle uploaded files, it will also process general POST data. All we have to do is add a listener for the `field` event to process forms containing both files and user data.

```
incoming.on('file', function (field, file) {
  response.write(file.name + ' received\n');
})
.on('field', function (field, value) {
  response.write(field + ': ' + value + '\n');
})
.on('end', function () {
  response.end('All files received');
});
```

There's no need to manually implement field data size limits as `formidable` takes care of this for us. However, we can change the defaults with `incoming.maxFieldsSize`, which allows us to limit the total byte count for the sum of all fields. This limit doesn't apply to file uploads.

Preserving filenames with formidable

When `formidable` places our files into the `uploads` directory, it assigns them a name consisting of a randomly generated hexadecimal number. This prevents files of the same name from being overwritten. But what if we want to know which files are which and yet still retain the unique filename advantage? We can alter the way `formidable` names each file during its `fileBegin` event as shown in the following code:

```
if (request.method === "POST") {
  var incoming = new formidable.IncomingForm();
  incoming.uploadDir = 'uploads';
  incoming.on('fileBegin', function (field, file) {
    if (file.name) {
      file.path += "-" + file.name;
    } //...rest of the code
  }).on('file', function (field, file) {
    //...rest of the code
  });
}
```

We've appended the original filename onto the end of the random filename assigned by `form.id`, separating them with a dash. Now we can easily identify our files. However, for many scenarios this may not be necessary as we would likely be outputting file information to a database and cross referencing it to randomly generated names.

Uploading via PUT

It's also possible to upload files via an HTTP PUT request. While we can only send one file per request, we don't need to do any parsing on the server side since the file will simply stream directly to our server, which means less server-side processing overhead. It would be magnificent if we could achieve this by changing our form's `method` attribute from POST to PUT but alas, no. However, thanks to the up and coming XMLHttpRequest Level 2 (xhr 2), we can now transfer binary data via JavaScript in some browsers (see <http://www.caniuse.com/#search=xmhttprequest%202>). We grab a file pointer using a `change` event listener on the input file element, then we open a PUT request and send the file. The following is for use in `form.html`, which we'll save as `put_upload_form.html`:

```
<form id="frm">
    <input type="file" id="userfile" name="userfile"><br>
    <input type="submit">
</form>
<script>
(function () {
    var userfile = document.getElementById('userfile'),
        frm = document.getElementById('frm'),
        file;
    userfile.addEventListener('change', function () {
        file = this.files[0];
    });
    frm.addEventListener('submit', function (e) {
        e.preventDefault();
        if (file) {
            var xhr = new XMLHttpRequest();
            xhr.file = file;
            xhr.open('put', window.location, true);
            xhr.setRequestHeader("x-uploadedfilename",
                file.fileName || file.name);
            xhr.send(file);
            file = '';
            frm.reset();
        }
    });
})();
</script>
```

`Id` is added to the form and file inputs while `method` and `enctype` attributes have been removed. We're using just one file element because we can only send one file per request, although the example could be extended to asynchronously stream multiple files to our server at once.

Our script attaches a `change` listener to the file input element. When the user selects a file we are able to capture a pointer to the file. As the form is submitted, we prevent default behavior, check if a file is selected, initialize an `xhr` object, open a PUT request to our server, set a custom header so we can grab the filename later, and send the file to our server. Our server looks like the following code:

```
var http = require('http');
var fs = require('fs');
var form = fs.readFileSync('put_upload.html');
http.createServer(function (request, response) {
  if (request.method === "PUT") {
    var fileData = new Buffer(+request.headers['content-length']);
    var bufferOffset = 0;
    request.on('data', function(chunk) {
      chunk.copy(fileData, bufferOffset);
      bufferOffset += chunk.length;
    }).on('end', function() {
      var rand = (Math.random()*Math.random())
        .toString(16).replace('.', '');
      var to = 'uploads/' + rand + "-" +
        request.headers['x-uploadedfilename'];
      fs.writeFile(to, fileData, function(err) {
        if (err) { throw err; }
        console.log('Saved file to ' + to);
        response.end();
      });
    });
  }
  if (request.method === "GET") {
    response.writeHead(200, {'Content-Type': 'text/html'});
    response.end(form);
  }
}).listen(8080);
```

Our PUT server follows a similar pattern to the simple POST server in the *Processing POST data* recipe. We listen to the data event and piece the chunks together. However, rather than string concatenate our data, we must throw our chunks into a buffer because a buffer can handle any data type including binary, whereas a string object will always coerce non-string data into string format. This changes the underlying binary resulting in corrupted files. Once the `end` event has triggered, we generate a random file name similar to the naming convention of `formidable` and write the file to our `uploads` folder.



This *Uploading via PUT* demonstration will not work in older browsers, so an alternative fall back should be provided in a production environment. Browsers that will support this method are IE 10 and above, Firefox, Chrome, Safari, iOS 5+ Safari, and Android browsers. However, due to browser vendors differing implementations of the same functionality, the example may need some tweaking for cross-browser compatibility.

See also

- ▶ *Sending email* discussed in Chapter 8, *Integrating Network Paradigms*
- ▶ *Using Node as an HTTP client* discussed in this chapter.

Using Node as an HTTP client

The HTTP object doesn't just provide server capabilities, it also affords us with client functionality. In this task, we're going to use `http.get` with `process` to fetch external web pages dynamically via the command line.

Getting ready

We are not creating a server, so in the naming convention we should use a different name for our new file, let's call it `fetch.js`.

How to do it...

`http.request` allows us to make requests of any kind (for example, GET, POST, DELETE, OPTION, and so on), but for GET requests we can use the short-hand `http.get` method as follows:

```
var http = require('http');
var urlOpts = {host: 'www.nodejs.org', path: '/', port: '80'};
http.get(urlOpts, function (response) {
  response.on('data', function (chunk) {
    console.log(chunk.toString());
  });
});
```

Essentially we're done.

```
node fetch.js
```

If we run the preceding command, our console will output the HTML of nodejs.org. However, let's pad it out a bit with some interactivity and error handling as shown in the following code:

```
var http = require('http');
var url = require('url');
var urlOpts = {host: 'www.nodejs.org', path: '/', port: '80'};
if (process.argv[2]) {
  if (!process.argv[2].match('http://')) {
    process.argv[2] = 'http://' + process.argv[2];
  }
  urlOpts = url.parse(process.argv[2]);
}
http.get(urlOpts, function (response) {
  response.on('data', function (chunk) {
    console.log(chunk.toString());
  });
}).on('error', function (e) {
  console.log('error:' + e.message);
});
```

Now we can use our script like this:

```
node fetch.js www.google.com
```

How it works...

`http.get` takes an object which defines the criteria of our desired request. We defined a variable called `urlOpts` for this purpose and set our host to `www.nodejs.org`. We use the `process.argv` property to check if a web address has been specified via the command line. Like `console`, `process` is a global variable that is always available within a Node runtime environment. `process.argv[2]` is the third command-line argument, with `node` and `fetch.js` being allocated to `[0]` and `[1]` respectively.

If `process.argv[2]` exists (that is, if an address has been specified) we append `http://`. If it isn't there (`url.parse` requires it), then replace the object in our default `urlOpts` with the output from `url.parse`. Happily, `url.parse` returns an object with the same properties `http.get` requires.

As a client, we are interacting with the server's response to us, rather than the client's request from us. So inside the `http.get` callback, we listen for the `data` event on `response` instead of (as with our server examples) `request`. As the `response` data stream arrives, we output the chunks to the console.

There's more...

Let's explore some of the possibilities of the underlying `http.request` method of `http.get`.

Sending POST requests

We'll need to fire up our `server.js` app from the *Processing POST data* recipe to receive our POST requests. Let's make a new file and call it `post.js`, which we'll use to send POST requests to our POST server.

```
var http = require('http');
var urlOpts = {host: 'localhost', path: '/',
    port: '8080', method: 'POST'};
var request = http.request(urlOpts, function (response) {
    response.on('data', function (chunk) {
        console.log(chunk.toString());
    });
}).on('error', function (e) {
    console.log('error:' + e.stack);
});
process.argv.forEach(function (postItem, index) {
    if (index > 1) { request.write(postItem + '\n'); }
});
request.end();
```

As we're using the more general `http.request`, we've had to define our HTTP verb in the `urlOpts` variable. Our `urlOpts` variable also specifies the server as `localhost:8080` (we must ensure that our POST server is running in order for this code to work).

As before, we set up an event listener in our callback for `data` on the `response` object. `http.request` returns a `client Request` object which we load into a variable called `request`. This is a newly declared variable, which holds the returned `client Request` object from our `http.request` method.

After our event listeners, we loop through the command-line arguments using the Ecmascript `5 for Each` method (which is safe to use in Node but not yet in browsers). In running this script, `node` and `post.js` would be the 0th and 1st arguments, so we check that our array `index` is greater than 1 before sending any arguments as POST data. We use `request.write` to send data similar to how we would use `response.write` if we were building a server. Even though it uses a callback, `for Each` is not asynchronous (it blocks until completion), so only after every element is processed is our POST data written and our request ended. This is how we use it:

```
node post.js foo=bar&x=y&anotherfield=anothervalue
```

Multipart file upload as a client

We'll use our upload server from *Handling File Uploads* to receive the files from our uploading client. To achieve this, we have to deal with the multipart data format. To inform a server of the client's intentions of sending multipart data, we set the `content-type` header to `multipart/form-data` with an additional attribute called `boundary`, which is a custom named delimiter, separating files in the multipart data.

```
var http = require('http');
var fs = require('fs');
var urlOpts = { host: 'localhost', path: '/', port: '8080', method: 'POST' };
var boundary = Date.now();
urlOpts.headers = {
  'Content-Type': 'multipart/form-data; boundary=' + boundary + '"'
};
```

We've required the `fs` module here too as we'll be needing that later to load our files.

We've set our `boundary` to the current Unix time (milliseconds since midnight, January 1, 1970). We won't need `boundary` again in this format, so let's update it with the required multipart double dash (--) prefix and set up our `http.request` call:

```
boundary = '--' + boundary;
var request = http.request(urlOpts, function (response) {
  response.on('data', function (chunk) {
    console.log(chunk.toString());
  });
}).on('error', function (e) {
  console.log('error:' + e.stack);
});
```

We want to be able to stream multipart data to the server, which may be compiled from multiple files. If we streamed these files while simultaneously attempting to compile them together into the multipart format, the data would likely be mashed together from different file streams in an unpredictable order becoming impossible to parse. So we need a way to preserve the data order.

We could build it all in one go and afterwards send it to the server. However, a more efficient (and Node-like) solution is to build the multipart message by progressively assembling each file into the multipart format as the file is streamed in, while instantly streaming the multipart data as it's being built.

To achieve this, we can use a self-iterating function, calling each recursion from within the `end` event callback to ensure each stream is captured separately and in order.

```
(function multipartAssembler(files) {
  var f = files.shift(), fSize = fs.statSync(f).size;
```

```
fs.createReadStream(f)
  .on('end', function () {
    if (files.length) { multipartAssembler(files); return; //early
finish}
    // any code placed here wont execute until no files are left
    // due to early return from function.
  });
}(process.argv.splice(2, process.argv.length)));
```

This is also a self-calling function because we've changed it from a declaration to an expression by wrapping parenthesis around it. Then we've called it by appending parenthesis, also passing in the command-line arguments, which specify what files to upload:

```
node upload.js file1 file2 fileN
```

We use `splice` on the `process.argv` array to remove the first two arguments (which would be `node` and `upload.js`). The result is passed into our `multipartAssembler` function as our `files` parameter.

Inside our function we immediately shift the first file off of the `files` array and load it into the variable `f`, which is passed into `createReadStream`. Once it's finished reading, we pass any remaining files back through our `multipartAssembler` function and repeat the process until the array is empty. Now let's flesh out our self-iterating function with multipart goodness as follows:

```
(function multipartAssembler(files) {
  var f = files.shift(), fSize = fs.statSync(f).size,
  progress = 0;
  fs.createReadStream(f)
    .once('open', function () {
      request.write(boundary + '\r\n' +
        'Content-Disposition: ' +
        'form-data; name="userfile"; filename=' + f +
        '\r\n' +
        'Content-Type: application/octet-stream\r\n' +
        'Content-Transfer-Encoding: binary\r\n\r\n');
    }).on('data', function(chunk) {
      request.write(chunk);
      progress += chunk.length;
      console.log(f + ': ' + Math.round((progress / fSize) *
10000)/100 + '%');
    }).on('end', function () {
      if (files.length) { multipartAssembler(files); return; //early
finish }
      request.end('\r\n' + boundary + '--\r\n\r\n\r\n');
    });
}(process.argv.splice(2, process.argv.length)));
```

We specify a part with the predefined boundary initially set in the `content-type` header. Each part needs to begin with a header, we latch on to the `open` event to send this header out.

`content-disposition` has three parts. In this scenario, the first part will always be `form-data`. The second part defines the name of the field (for instance, the `name` attribute of a file input), and the original filename. The `content-type` can be set to whatever mime is relevant. However, by setting all files to `application/octet-stream` and `content-transfer-encoding` to `binary`, we can safely treat all files the same way if all we're doing is saving to disk without any interim processing. We finish each multipart header with a double CRLF (`\r\n\r\n`) at the end of our `request.write`.

Also, notice we've assigned a new `progress` variable at the top of the `multipartAssembler` function. We use this to determine the relative percent of the upload by dividing the chunks received so far (`progress`), by the total file size (`fSize`). This calculation is performed in our `data` event callback, where we also stream each chunk to the server.

In our `end` event, if there are no more files to process, we end the request with the final multipart boundary which is the same as other boundary partitions except it has leading and trailing slashes.

See also

- ▶ *Working with real data: fetching trending tweets* discussed in *Chapter 3, Working with Data Serialization*

Implementing download throttling

For incoming streams, Node provides `pause` and `resume` methods, but not so for outbound streams. Essentially, this means we can easily throttle upload speeds in Node but download throttling requires a more creative solution.

Getting ready

We'll need a new `server.js` along with a good-sized file to serve. With the `dd` command-line program, we can generate a file for testing purposes.

```
dd if=/dev/zero of=50meg count=50 bs=1048576
```

This will create a 50 MB file named `50meg` which we'll be serving.



For a similar Windows tool that can be used to generate a large file, check out <http://www.bertel.de/software/rdfc/index-en.html>.

How to do it...

To keep things as simple as possible our download server will serve just one file, but we'll implement it in a way which would allow us to easily plug in some router code to serve multiple files. First, we will require our modules and set up an `options` object for file and speed settings.

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

var options = {}
options.file = '50meg';
options.fileSize = fs.statSync(options.file).size;
options.kbps = 32;
```

If we were serving multiple files, our `options` object would be largely redundant. However, we're using it here to emulate the concept of a user-determined file choice. In a multifile situation, we would be loading file specifics based upon the requested URL instead.



To see how this recipe could be configured to serve and throttle more than one file, check out the routing recipes in *Chapter 1, Making a Web Server*.

The `http` module is for the server while the `fs` module is for creating a `readStream` and grabbing the size of our file.

We're going to restrict how much data is sent out at once, but we first need to get the data in. So let's create our server and initialize a `readStream`

```
http.createServer(function(request, response) {
  var download = Object.create(options);
  download.chunks = new Buffer(download.fileSize);
  download.bufferOffset = 0;

  response.writeHead(200, {'Content-Length': options.fileSize});

  fs.createReadStream(options.file)
    .on('data', function(chunk) {
      chunk.copy(download.chunks, download.bufferOffset);
      download.bufferOffset += chunk.length;
    })
    .once('open', function() {
      // this is where the throttling will happen
    });
}).listen(8080);
```

We've created our server and specified a new object called `download`, which inherits from our `options` object. We add two properties to our request-bound `download` object: a `chunks` property that collects the file chunks inside the `readStreamdata` event listener and a `bufferOffset` property that will be used to keep track of the amount of bytes loaded from disk.

All we have to do now is the actual throttling. To achieve this, we simply apportion out the specified number of kilobytes from our buffer every second, thus achieving the specified kilobytes per second. We'll make a function for this, which will be placed outside of `http.createServer` and we'll call our function `throttle`.

```
function throttle(download, cb) {
  var chunkOutSize = download.kbps * 1024,
    timer = 0;

  (function loop(bytesSent) {
    var remainingOffset;
    if (!download.aborted) {
      setTimeout(function () {
        var bytesOut = bytesSent + chunkOutSize;

        if (download.bufferOffset > bytesOut) {
          timer = 1000;
          cb(download.chunks.slice(bytesSent, bytesOut));
          loop(bytesOut);
          return;
        }

        if (bytesOut >= download.chunks.length) {
          remainingOffset = download.chunks.length - bytesSent;
          cb(download.chunks.slice(remainingOffset, bytesSent));
          return;
        }

        loop(bytesSent); // continue to loop, wait for enough data
      }, timer);
    }
  })(0));

  return function () { // return a function to handle an abort
    scenario
    download.aborted = true;
  };
}
```

`t hr ot t l e` interacts with the `downl oad` object created on each server request to measure out each chunk according to our predetermined `opt i ons . kbps` speed. For the second parameter (`c b`), `t hr ot t l e` accepts a functional callback. `c b` in turn takes one parameter, which is the chunk of data that `t hr ot t l e` has determined to send. Our `t hr ot t l e` function returns a convenience function that can be used to end the loop on abort, avoiding infinite looping. We initialize download throttling by calling our `t hr ot t l e` function in the server callback when the `readSt ream` opens.

```
// ... previous code
fs.createReadStream(options.file)
  .on('data', function (chunk) {
    chunk.copy(downl oad.chunks, downl oad.bufferOffset);
    downl oad.bufferOffset += chunk.length;
  })
  .once('open', function () {
    var handleAbort = t hr ot t l e(downl oad, function (send) {
      response.write(send);
    });

    request.on('close', function () {
      handleAbort();
    });
  });
});

}).listen(8080);
```

How it works...

The key to this recipe is our `t hr ot t l e` function. Let's walk through it. To achieve the specified speed, we send a chunk of data of a certain size every second. The size is determined by the desired amount of kilobytes per second. So, if `downl oad . kbps` is 32, we'll send 32 KB chunks every second.

Buffers work in bytes, so we set a new variable called `chunkOut Si ze` and multiply `downl oad . kbps` by 1024 to realize the appropriate chunk size in bytes. Next, we set a `ti mer` variable which is passed into `set Ti meout`. It is first set to 0 on two accounts. For one, it eliminates an unnecessary initial 1000 millisecond overhead, allowing our server the opportunity to immediately send the first chunk of data, if available. Secondly, if the `downl oad . chunks` buffer is not full enough to accommodate the demand of `chunkOut Si ze`, the embedded `loop` function recurses without changing `ti mer`. This causes the CPU to cycle in real time until the buffer loads enough data to deliver a whole chunk (a process which should take less than a second).

Once we have enough data for the first chunk, `timer` is set to 1000 because from here on out we want to push a chunk every second.

`loop` is the guts of our throttling engine. It's a self-recursive function which calls itself with one parameter: `bytesSent`. The `bytesSent` parameter allows us to keep track of how much data has been sent so far, and we use it to determine which bytes to slice out of our `download.chunks` buffer using `Buffer.slice`. `Buffer.slice` takes two parameters, `start` and `end`. These two parameters are fulfilled with `bytesSent` and `bytesOut` respectively. `bytesOut` is also used against `download.bufferOffset` to ensure we have enough data loaded for a whole chunk to be sent out.

If there is enough data, we proceed to set the `timer` to 1000 to initiate our chunk per second policy, then pass the result of `download.chunks.slice` into `cb` which becomes our `send` parameter.

Back inside our server, our `send` parameter is passed to `response.write` within our `throttle` callback, so each chunk is streamed to the client. Once we've passed our sliced chunk to `cb` we call `loop(bytesOut)` for a new iteration (thus `bytesOut` transforms into `bytesSent`), then we return from the function to prevent any further execution.

The third and final place `bytesOut` appears is in the second conditional statement of the `set Timeout` callback, where we use it against `download.chunks.length`. This is important for handling the last chunk of data. We don't want to loop again after the final chunk has been sent, and if `options.kbps` doesn't divide exactly into the total file size, the final `bytesOut` would be larger than the size of the buffer. If passed into the `slice` method unchecked, this would cause an object out of bounds (`oob`) error.

So if `bytesOut` equals, or is greater than, the memory allocated to the `download.chunks` buffer (that is, the size of our file), we `slice` the remaining bytes from our `download.chunks` buffer and return from the function without calling `loop`, effectively terminating recursion.

To prevent infinite looping when the connection is closed unexpectedly (for instance during connection failure or client abort) `throttle` returns another function, which is caught in the `handleAbort` variable and called in the `close` event of `response`. The function simply adds a property to the `download` object to say the download has been aborted. This is checked on each recursion of the `loop` function. As long as `download.aborted` isn't `true` it continues to iterate, otherwise the looping stops short.

[ There are (configurable) limits on operating systems defining how many files can be opened at once. We would probably want to implement caching in a production download server to optimize file system access. For file limits on Unix systems, see <http://www.stackoverflow.com/questions/34588/how-do-i-change-the-number-of-open-files-limit-in-linux>.]

Enabling resumes from broken downloads

If a connection breaks, or a user accidentally aborts a download, the client may initiate a resume request by sending a Range HTTP header to the server. A Range header would look something like this:

```
Range: bytes=512-1024
```

When a server agrees to handle a Range header, it sends a 206 Partial Content status and adds a Content-Range header in the response. Where the entire file is 1 MB, a Content-Range reply to the preceding Range header might look as follows:

```
Content-Range: bytes 512-1024/1024
```

Notice that there is no equals sign (=) after bytes in a Content-Range header. We can pass an object into the second parameter of `fs.createReadStream`, which specifies where to start and end reading. Since we are simply handling resumes, we only need to set the `start` property.

```
// requires, options object, throttle function, create server etc...
download.readStreamOptions = {};
download.headers = {'Content-Length': download.fileSize};
download.statusCode = 200;
if (request.headers.range) {
    download.start = request.headers.range.replace('bytes=','').
split('-')[0];
    download.readStreamOptions = {start: +download.start};
    download.headers['Content-Range'] = "bytes " + download.
start + "-" + download.fileSize + "/" +
download.fileSize;
    download.statusCode = 206; // partial content
}
response.writeHead(download.statusCode, download.headers);
fs.createReadStream(download.file, download.readStreamOptions)
//...rest of the code....
```

By adding some properties to `download`, and using them to conditionally respond to a Range header, we can now handle resume requests.

See also

- ▶ *Setting up a router* discussed in *Chapter 1, Making a Web Server*
- ▶ *Caching content in memory for immediate delivery* discussed in *Chapter 1, Making a Web Server*
- ▶ *Communicating via TCP* discussed in *Chapter 8, Integrating Networking Paradigms*

3

Working with Data Serialization

In this chapter we will cover:

- ▶ Converting an object to JSON and back again
- ▶ Converting an object to XML and back again
- ▶ Browser-server transmission via AJAX
- ▶ Working with real data: fetching trending tweets

Introduction

If we want to give third parties safe access to raw data, we can use serialization to send it in a format that the requester will understand. In this chapter, we'll be looking at data serialization in two well-known standards, JSON and XML.

Converting an object to JSON and back again

JSON (JavaScript Object Notation) is very closely related to JavaScript objects because it's a subset of JavaScript. This task will demonstrate how to use the building blocks of JSON conversion: `JSON.parse` and `JSON.stringify`.

Getting ready

We'll need to create two new files called `profiles.js` and `json_and_back.js`.

How to do it...

Let's create the object that we'll later be converting to JSON.

```
module.exports = {
    ryan: {
        name: "Ryan Dahl",
        irc: "ryah",
        twitter: "ryah",
        github: "ry",
        location: "San Francisco, USA",
        description: "Creator of node.js"
    },
    isaac: {
        name: "Isaac Schlueter",
        irc: "isaacs",
        twitter: "izs",
        github: "isaacs",
        location: "San Francisco, USA",
        description: "Author of npm core contributor"
    },
    bert: {
        name: "Bert Belder",
        irc: "piscisauereus",
        twitter: "piscisauereus",
        github: "piscisauereus",
        location: "Netherlands",
        description: "Windows support, overall contributor"
    },
    tj: {
        name: "TJ Holowaychuk",
        irc: "tjholowaychuk",
        twitter: "tjholowaychuk",
        github: "visionmedia",
        location: "Victoria, BC, Canada",
        description: "Author of express, jade and other popular modules"
    },
    felix: {
        name: "Felix Geisendorfer",
        irc: "felixgeisendorfer",
        twitter: "felixgeisendorfer",
        github: "felixgeisendorfer",
        location: "Berlin, Germany",
        description: "Author of ko, a minimalist library for working with streams"
    }
};
```

```

    irc: "felixge",
    twitter: "felixge",
    github: "felixge",
    location: "Berlin, Germany",
    description: "Author of formidable, active core developer"
}
};

```

This object contains profile information on some of the leading members of the Node Community (though it's entirely non-exhaustive and doesn't even contain all of the core development team). One thing to note here is the use of `module.exports`. We'll be seeing more of this in *Chapter 9, Writing Your Own Module*. We're using `module.exports` to modularize our `profiles` object here in a bid to keep our code uncluttered. We can load any expression into `module.exports`, save it as a separate file (which in our case, we'll call `profiles.js`), and use `require` in our main file to dynamically load it at initialization.

```
var profiles = require('./profiles'); // note the .js suffix is optional
```

Nice and tidy. To convert our `profiles` object into a JSON representation, we use `JSON.stringify`, which will return a string composed of JSON data. We're going to fundamentally alter our object (which is now a string) using `replace`.

```
profiles = JSON.stringify(profiles).replace(/name/g, 'fullname');
```

Here we have called `replace`, using a regular expression with the global `g` option to change every occurrence of `name` in our JSON string to `fullname`.

But wait! There appears to be some kind of mistake. Felix's last name is missing an umlaut! Let's correct it by converting our JSON data back into an object, and correct his name by altering the value of the re-designated `fullname` property:

```
profiles = JSON.parse(profiles);
profiles.felix.fullname = "Felix Geisendorfer";
console.log(profiles.felix);
```

When we run our application, `console.log` will output the following:

```
{
  fullname: 'Felix Geisendorfer',
  irc: 'felixge',
  twitter: 'felixge',
  github: 'felixge',
  location: 'Berlin, Germany',
  description: 'Author of formidable, active core developer' }
```

The first key is now `fullname`, and `Geisendorfer` is spelled correctly.

How it works...

First, we have an everyday JavaScript object, which we serialize into a JSON representation. We also call the `String.replace` method on our JSON string, changing every occurrence of `name` into `full name`.

Using `replace` in this way and context isn't an advisable practice since any occurrences of `name` are replaced. There could very easily have been other places in the string where `name` may have existed, which would be replaced unintentionally. We used `replace` here to affirm that profiles have become a JSON string, as we couldn't use `replace` on an object.

Then we convert our modified JSON string back into an object, using `JSON.parse`. To test that our keys were indeed transformed from `name` to `full name`, and to affirm that we are again working with an object, we correct the `felix` profile via `profiles.felix.fullname`, and then log `profiles.felix` to the console.

There's more...

JSON is a highly versatile and flexible tool for cross-platform communication. Let's look at a more advanced application of the standard.

Constructing JSONP responses

JSONP (JSON with Padding) is a cross-domain policy workaround that allows developers to interface with resources on other domains. It involves defining a callback function on the client side that handles JSON via its first parameter, then passing the name of this callback as a query argument in the `src` attribute of a `script` element, which points to a web service on another domain. The web service then returns the JSON data, wrapped in a function named according to the query argument set client side. It's possibly easier to illustrate this in code.

```
<html>
<head>
<script>
  var who = 'ryan';
  function cb(o) {
    alert(o.name + ' : ' + o.description);
  }
  var s = document.createElement('script');
  s.src = 'http://localhost:8080/?callback=cb&who=' + who;
  document.getElementsByTagName("head")[0].appendChild(s);
</script>
</head>
</html>
```

We define a function called `cb` which takes an object as its parameter, then we output the `name` and `description` properties. Prior to this, we set a variable called `who` which will be passed to the server to grab specific data for us. We then dynamically inject a new script element, setting `src` to a figurative third-party domain (which for easy demonstration is `localhost`) and adding `callback` and `who` query arguments. The value of `callback` matches the name of our function `cb` function. Our server uses this parameter to wrap JSON in a function invocation.

```
var http = require('http');
var url = require('url');
var profiles = require('./profiles');

http.createServer(function (request, response) {
  var urlObj = url.parse(request.url, true),
    cb = urlObj.query.callback, who = urlObj.query.who,
    profile;

  if (cb && who) {
    profile = cb + "(" + JSON.stringify(profiles[who]) + ")";
    response.end(profile);
  }
}).listen(8080);
```

We create a server, extract the `callback` and `who` query parameters, and write a response containing a function call passing our JSON data in as its parameter. This script is loaded by our client, where the `cb` function is called and JSON is received into the function as an object (because it looks like one).

Security and JSONP

Since JSONP uses script injection, any script could be inserted into our page. Therefore, it's highly recommended that this method only be used with trusted sources. An untrusted source could run evil code on the page.

See also

- ▶ *Browser-server transmission via AJAX* discussed in this chapter
- ▶ *Working with real data: fetching trending tweets* discussed in this chapter

Converting an object to XML and back again

Since JSON is a string-based representation of a JavaScript object, converting between the two is straightforward. However, XML is less convenient to work with. Nevertheless, there may be times we have to work with it, for instance, if an API works only in XML or if we were contracted with a project that specifies XML support.

There are various non-core XML parsers available. One such parser is the non-core module `xml2js`. The premise of `xml2js` is that working with objects in JavaScript is more suitable than working with XML. `xml2js` provides a basis for us to interact with XML by converting it to a JavaScript object.

In this task, we're going to write a function that uses our `profiles` object featured in the previous recipe to create a valid XML string, which we'll then push through `xml2js`, thus converting it back into an object.

Getting ready

Before we start, let's create our file `xml_and_back.js`, making sure we have our separately modularized `profiles.js` in the same directory. We should also install `xml2js`.

```
npm install xml2js
```

How to do it...

To begin with, we'll need to require our `profiles` object along with `xml2js`:

```
var profiles = require('./profiles');
var xml2js = new (require('xml2js')).Parser();
```

Notice that rather than just requiring the `xml2js` module, we've initialized a new instance of its `Parser` method and loaded that as our `xml2js` variable. This is to do with the way the `xml2js` module works. We have to create a new `Parser` instance in order to parse a piece of XML into an object. Since our code is relatively simple we may as well do the initialization work while we require it.

Just as XML has a tree-like structure, objects can have objects nested within them. We need a function that can loop through our object and all sub-objects, converting all properties into parent XML nodes, and all non-object values into text XML nodes:

```
function buildXml(rootObj, rootName) {
  var xml = "<?xml version='1.0' encoding='UTF-8'?>\n";
  rootName = rootName || 'xml';
  xml += "<" + rootName + ">\n";
  (function traverse(obj) {
```

```

Object.keys(obj).forEach(function(key) {
  var open = "<" + key + ">",
      close = "</ " + key + ">\n",
      isTxt = (obj[key]
        && {} .toString.call(obj[key]) !== "[object Object]");

  xml += open;

  if (isTxt) {
    xml += obj[key];
    xml += close;
    return;
  }

  xml += "\n";
  traverse(obj[key]);
  xml += close;
});

})(rootObj);

xml += "</ " + rootName + ">";
return xml;
}

```

`buildXml` takes two parameters, the object and a string to name the first root XML node, and returns a string of XML data representing our object.

Let's replace all occurrences of `name` with `fullName` as in our *Converting an object to JSON and back again* recipe.

```

profiles = buildXml(profiles, 'profiles').replace(/name/g,
'fullName');
console.log(profiles); // <-- show me the XML!

```

Now we'll turn `profiles` back into an object, correct Felix Geisendorfer's name using the renamed `fullName` property, then log `Felix` to console to show it has worked.

```

xml2js.parseString(profiles, function(err, obj) {
  profiles = obj;
  profiles.felix.fullName = "Felix Geisendorfer";
  console.log(profiles.felix);
});

```

`xml2js.parseString` takes the XML (which at this point is held in the `profiles` variable) and assembles it into an object which is passed through as the `obj` parameter in its callback.

How it works...

A JavaScript object is a key value store, whereas XML is a resource-centric markup language. In XML, keys and values could be represented in two ways: either as a parent node and a child node or as attributes on an XML node. We converted our keys and values to parent and child nodes, mainly because a single XML node is filled with lots of attributes, while valid XML, seems to violate the spirit of XML.

We achieved our conversion with `buidXml`, which is a function wrapping another self-calling recursive function named `traverse`. We do this to take advantage of the closure principle in JavaScript which allows us to share variables between inner and outer function. This enables us to use the outer `xml` variable to assemble our serialized XML.

In our outer function, we begin with the `<?xml ?>` declaration setting the required `version` attribute and the optional `encoding` attribute to UTF-8. We also wrap any output that `traverse` renders in an ending and closing tag named after our `rootName` parameter. So in our case, `buidXml` puts the following into our `xml` variable:

```
<?xml version='1.0' encoding='UTF-8' ?>
<profiles>
    <!!-- Traverse XML Out put Here -->
</profiles>
```

If `rootName` is missing, we default to `<xml>` as the root node. Our `traverse` inner function takes a single parameter, which is the object to be converted to XML. We pass in `rootObj` to the calling parentheses:

```
(function traverse(obj) {
    // traverse function code here...
    }(rootObj)); // passing in our root object parameter
```

`traverse` uses `for Each` to loop through the keys of this object, accessing each key via the first parameter of the `for Each` callback. We use the name of each key to generate the opening and closing of XML tags, and append the open tag to our shared `xml` variable. Then we check our `isTxt` variable, which tests for a nested object and returns `true` if it's not an object (assuming it must therefore be text). If `isTxt` is `true`, we output the value of our current property and return from the `for Each` callback, continuing onto the next property. This is how we get our text nodes—the values. Otherwise, we append a line feed to `xml` and call `traverse` on the sub-object, going through the exact same process, only this time it's embedded in the parent `traverse` function. Once our nested call to `traverse` has returned, we append the `close` tag to `xml` and our `traverse` function is complete. Finally, our outer function appends the closing root node tag and returns all the generated XML.

There's more...

We can adapt our code further to better integrate with the `xml2js` library by mirroring its interpretations of certain XML features into JavaScript object equivalents. We can also extend it to translate more complex JavaScript objects into valid XML.

Objects containing arrays and functions

As well as objects and strings, object properties can contain functions and arrays. As it stands, our recipe will interpret these as text, outputting a comma-separated list of values where arrays are concerned, and returning the contents of functions within text nodes.

This isn't ideal, so we'll alter our `traverse` function to handle these types:

```
(function traverse(obj) {
  Object.keys(obj).forEach(function (key) {
    var open = "<" + key + ">",
        close = "</ " + key + ">\n",
        nonObj = (obj[key]

          && {}.toString.call(obj[key]) !== "[object Object]"),
        isArray = Array.isArray(obj[key]),
        isFunc = (typeof obj[key] === "function");

    if (isArray) {
      obj[key].forEach(function (xmlNode) {
        var childNode = {};
        childNode[key] = xmlNode;
        traverse(childNode);
      });
      return;
    }

    xml += open;
    if (nonObj) {
      xml += (isFunc) ? obj[key]() : obj[key];
      xml += close;
      return;
    }
  })
  //rest of traverse function
})
```

We'll save our altered code as `xml_with_arrays_and_functions.js`. For semantics' sake, we renamed `isTxt` to `nonObj` and added two more tester variables, `isArray` and `isFunc`. If the value of the object we are traversing is an array, we create a temporary `childNode` object which we pass back into `traverse`. We do the same for each value of the array, each time creating a new `childNode` object where the key is the same but the value is the next array element. This effectively creates multiple child nodes of the same name.

To test the array support, let's copy our `profiles.js` file to `profiles_with_arrays_and_functions.js`, requiring it instead of `profiles.js`. Ryan Dahl also pushes to another Github account: joyent. So let's update his profile with an array of Github accounts:

```
module.exports = {
  ryan: {
    name: "Ryan Dahl",
    irc: "ryah",
    twitter: "ryah",
    github: ["ry", "joyent"],
    location: "San Francisco, USA",
    description: "Creator of node.js"
  },
  //...rest of profiles...
```

Now if we do the following:

```
profiles = buildXml(profiles, 'profiles');
console.log(profiles); // <-- show me the XML!
```

And take a look at the output, we'll see that Ryan has two Github XML nodes:

```
<?xml encoding='UTF-8'?>
<profiles>
<ryan>
<name>Ryan Dahl</name>
<irc>ryah</irc>
<twitter>ryah</twitter>
<github>ry</github>
<github>joyent</github>
<location>San Francisco, USA</location>
<description>Creator of node.js</description>
</ryan>
<!!-- REST OF THE XML OUTPUT -->
```

Our other variable, `isFunc`, is checked within the `nonObj` conditional statement. We use it to determine whether we should just add the text of our object property to our `xml` variable, or else call the object property to receive its return value. Bert's IRC, Twitter, and Github accounts are all the same, so let's add methods that pull the IRC and Twitter values from his Github value:

```
///...prior profiles code.
bert: {
  name: "Bert Belder",
  irc: function () { return this.github; },
  twitter: function () { return this.github; },
  github: "piscisaureus",
```

```

        location: "Netherlands",
        description: "Windows support, overall contributor"
    },
//...rest of profiles code...

```

If we build our XML from our object and then convert it back to an object using `xml2js`, these properties should no longer be functions, but instead be the returned values of the functions/methods:

```

xml2js.parseString(profiles, function (err, obj) {
  profiles = obj;
  console.log(profiles.bert);
});

```

And the output will be as follows:

```

{ name: 'Bert Belder',
  irc: 'piscisaureus',
  twitter: 'piscisaureus',
  github: 'piscisaureus',
  location: 'Netherlands',
  description: 'Windows support, overall contributor' }

```

Generating XML attributes

In XML, we can either express data relationships with parent, child, and text nodes, or by using attributes. If we want to include the ability of our `buildXml` function to cater to XML attributes, we need a convention for defining attributes within our objects. When converting from XML to objects, `xml2js` interprets attributes by adding a property containing an object that has a special `@property`, which in turn holds another sub-object containing the attributes. By implementing the same convention in `buildXml`, we can make our code work nicely alongside `xml2js`. Let's take the `profiles` object in `profiles_with_arrays_and_functions.js` and further update the `location` properties as follows:

```

module.exports = {
  ryan : {
    //ryans other keys here
    location: {'@': {city: 'San Francisco', country: 'USA'}},
    description: "Creator of node.js"
  },
  isaac : {
    //isaacs other keys here
    location: {'@': {city: 'San Francisco', country: 'USA'}},
    description: "Author of npm, core contributor"
  },
  bert : {

```

```
// berts other keys here...
location: {'@: {country: 'Netherlands'},,
description: "Windows support, overall contributor"
},
tj : {}, //-- TJ's keys
felix : {} //-- Felix's keys
}
```

We'll save this as `profiles_with_attributes.js`, and alter the `require` location for the `profiles` variable in our `xml_and_back_with_arrays_and_functions.js` code, saving as `xml_and_back_with_attributes.js`:

```
var profiles = require('../profiles_with_attributes');
```

Let's write another function that should be placed inside the `buildXml` function to handle our attributes:

```
function attributes(obj, key) {
  if (obj[key].hasOwnProperty('@')) {
    xml = xml.substr(0, xml.length - 1); // remove the ">" part of
    open tag

    Object.keys(obj[key]['@']).forEach(function (attrKey) {
      xml += ' ' + attrKey + '=' + obj[key]['@'][attrKey] + '';
    });

    xml += ">"; // add the ">" back on

    delete obj[key]['@']; // remove the key so it isn't traversed as
    an object
  }
}
```

Our new `attributes` function should be placed within our `buildXml` function and will be called inside `traverse`, just after we add our key's open tag variable to the `xml` variable, and just before we check for `nonObj` nodes:

```
(function traverse(obj) {
  //... prior traverse function code...
  xml += open;
  attributes(obj, key);
  If (nonObj) {
    //rest of traverse function code...
  }
})
```

We pass in the object and key currently being handled by our `traverse` function, which checks if this particular property of `obj` contains a property named `@`. We're also implicitly checking if the value of our current object key is in itself an object as only objects have properties.

The current attribute `@property` corresponds to the current tag. So if an `@property` is found, we remove the last character of `xml` (which would be a right angle bracket `>`) and loop through the keys of our sub-object (`obj[key][@]`) adding each key and its value to the last open tag to be appended to `xml`, re-appending the right angle bracket when done. If we left the `@object` inside our `profiles` object, it would later be passed back into the `traverse` function causing the following behavior:

```
<@>
<city>San Francisco</city>
<country>USA</country>
</@>
```

We don't want that, so we finish off by removing the `attributes` sub-object from our object. Underneath our `buildXml` function we have the following code:

```
profiles = buildXml(profiles, 'profiles').replace(/name/g,
'fullName');
console.log(profiles); // show me the XML!
```

This changes `name` keys to `fullName` and outputs our XML to console, rendering `location` tags with attributes like so:

```
<ryan>
<fullName>Ryan Dahl </fullName>
<irc>ryah</irc>
<twitter>ryah</twitter>
<github>ry</github>
<github>joyent</github>
<location city="San Francisco" country="USA">
</location>
<description>Creator of node.js</description>
</ryan>
<!-- rest of the XML output -->
```

Text values alongside attribute declarations

Our attribute solution uncovers another issue. There isn't a way for attributed nodes to contain text nodes, since we convert string types to text nodes but use an object to declare attributes. `xml2js` answers the antipathy of this problem with the `charkey` property. With the following code we become fully compatible with `xml2js`:

```
// previous code
if (key === '#') { // explicit text
  xml += obj[key] + '\n';
  return;
}
xml += open;
```

```
    attributes(obj, key);
    if (nonObj) {
        //rest of the code
```

Now the dilemma is resolved and we can explicitly add attributed nodes which contain text nodes, like so:

```
// prior profiles
tj : {
    name: "TJ Holowaychuk",
    irc: "tjholowaychuk",
    twitter: "tjholowaychuk",
    github: "visionmedia",
    location: {'@': {city: 'Victoria', country: 'Canada'}, region:
        {'#': 'British Columbia', '@': {type: 'province'}}},
    description: "Author of express, jade and other popular modules"
},
//rest of profiles
```

Which renders:

```
<irc>tjholowaychuk</irc>
<twitter>tjholowaychuk</twitter>
<github>visionmedia</github>
<github s="special">
</github>
<location city="Victoria" country="Canada">
<region type="province">
    British Columbia
</region>
</location>
<description>Author of express, jade and other popular modules</
description>
</tj>
```

See also

- ▶ *Converting an object to JSON and back again* discussed in this chapter
- ▶ *Browser-server transmission via AJAX* discussed in this chapter
- ▶ *Working with real data: fetching trending tweets* discussed in this chapter

Browser-server transmission via AJAX

We can enhance the user experience by loading new content directly into the page via AJAX, rather than loading a new page for each content request.

In this recipe, we're going to transfer our serialized data to the browser as the user requests it and then interact with our client-side data. We'll implement a profile viewer in the browser, which retrieves a selected profile in either JSON or XML, outputting the key-values or parent-child nodes for that profile.

Getting ready

We're going to continue to work with our `profiles.js` object module (from the first two recipes of this chapter). For XML delivery, we'll also grab our `buildXml` function from the *Converting an object to XML and back again* recipe, converting it to a simple module (just like we did with our `profiles` object in the previous recipe):

```
module.exports = function buildXml(rootObj, rootName) {
  // ... buildXml function code
}
```

We'll save this as `buildXml.js` and place it in a folder with a copy of our `profiles.js` file, along with two newly created files: `server.js` and `index.html`.

How to do it...

Let's start with our `index.html` file. We'll quickly implement a rough layout for our profile viewer consisting of a `form` with two `select` elements, a `div` for outputting formatted object data, and a `text area` element for presenting the raw serialized data.

```
<!DOCTYPE html>
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js">
</script>
<style>
#frm, #raw {display: block; float: left; width: 210px}
#raw {height: 150px; width: 310px; margin-left: 0.5em}
</style>
</head>
<body>
<form id="frm">
Profile: <select id="profiles">
```

```
<option></option>
</select> <br>
For mat : <select id=format>
    <option value=json> JSON </option>
    <option value=xml> XML </option>
</select><br> <br>
<div id=output></div>
</form>
<textarea id=raw></textarea>
</body>
</html>
```

Notice that we've included jQuery to obtain cross-browser benefits, particularly in the area of AJAX requests. We'll be utilizing jQuery in our client-side script shortly, but first let's make our server.

For our modules, we'll need `http`, `path`, and `fs` along with our custom `profiles` and `buildXml` modules. For our code to work, we'll need to have `index.html` hosted within our server in order to prevent cross-origin policy errors.

```
var http = require('http');
var fs = require('fs');
var path = require('path');
var profiles = require('../profiles');
var buildXml = require('../buildXml');

var index = fs.readFileSync('index.html');
var routes,
    mimes = {xml: "application/xml", json: "application/json"};
```

We've also defined `routes` and `mimes` variables so we can answer requests for specific data from the client along with the correct `Content-Type` header. We'll create two routes, one will deliver a list of profile names, the other will indicate a request for a particular profile.

```
routes = {
  'profiles': function (format) {
    return output(Object.keys(profiles), format);
  },
  '/profile': function (format, basename) {
    return output(profiles[basename], format, basename);
  }
};
```

Our `output` function, which we just referred to in `routes`, should be placed above the `routes` object and looks like the following code:

```
function output(content, format, rootNode) {
    if (!format || format === 'json') {
        return JSON.stringify(content);
    }
    if (format === 'xml') {
        return buildXml(content, rootNode);
    }
}
```

To finish our server, we simply call `http.createServer` and interact with our `routes` object inside the callback, outputting `index.html` where no routes are found:

```
http.createServer(function (request, response) {
    var dirname = path.dirname(request.url),
        extname = path.extname(request.url),
        basename = path.basename(request.url, extname);

    extname = extname.replace('.','');
    // remove period

    response.setHeader("Content-Type", mimes[extname] || 'text/html');

    if (routes.hasOwnProperty(dirname)) {
        response.end(routes[dirname](extname, basename));
        return;
    }

    if (routes.hasOwnProperty(basename)) {
        response.end(routes[basename](extname));
        return;
    }

    response.end(index);

}).listen(8080);
```

Finally, we need to write our client-side code to interface with our server over AJAX, which is to be placed in script tags just underneath our `#rawtextarea`, but above the closing `</body>` tag (to ensure the HTML elements have loaded before script execution) of our `index.html` file:

```
<script>
$.get('http://localhost:8080/profiles',
    function (profile_names) {
        $.each(profile_names, function (i, pname) {
```

```
$(' #profiles').append('<option>' + pname + '</option>');
    });
}, 'json');
$(' #formats, #profiles').change(function () {
    var format = $(' #formats').val();
    $.get('http://localhost:8080/profile/' + $(' #profiles').val() + '.' +
format,
        function (profile, stat, jqXHR) {
            var cT = jqXHR.getResponseHeader('Content-Type');
            $(' #raw').val(profile);
            $(' #output').html('');
            if (cT === 'application/json') {
                $.each($.parseJSON(profile), function (k, v) {
                    $(' #output').append('<b>' + k + '</b> : ' + v + '<br>');
                });
                return;
            }

            if (cT === 'application/xml') {
                profile = jqXHR.responseXML.firstChild.childNodes;
                $.each(profile,
                    function (k, v) {
                        if (v && v.nodeType === 1) {
                            $(' #output').append('<b>' + v.tagName + '</b> : ' +
v.textContent + '<br>');
                        }
                    });
            }

        },
        'text');
});
</script>
```

How it works...

Let's begin with the server. Inside our `http.createServer` callback, we set the appropriate header and check to see if the `routes` object has the specified directory name. If the directory name exists in `routes`, we call it as a function passing in `basename` and `extname` (we use `extname` to determine the desired format). In cases where there is no directory name match, we check for an existing property matching `basename`. If there is one, we call it and pass in the extension (if any). If both tests turn out to be false, we simply output the contents of our `index.html` file.

Our two routes are `profiles` and `/profile`, the latter has a preceding slash which corresponds to the way `path.dirname` returns the directory name of a path. Our `/profile` route is designed to allow for a sub-path containing the requested profile and format. For instance, `http://localhost:8080/profile/ryan.json` will return Ryan's profile in JSON format (if no extension is given, we default to JSON format).

Both the `profiles` and `/profile` methods utilize our custom `output` function which, using the `format` parameter (originally `extname` in the `http.createServer` callback), generates either JSON (using `JSON.stringify`) or XML (with our very own `buildXml` function) from the `content` passed to it. `output` also takes a conditional third parameter, which is passed along to `buildXml` to define the root Node of the generated XML.

On the client side, the first thing we do is call the jQuery `$.get` method for `http://localhost:8080/profiles`. This causes the server to call the `profiles` method on the `route` object. This in turn calls our `output` function with an array of top-level properties from our `profiles.js` object. Since we didn't specify an extension in `$.get`, the `output` function will default to JSON format and deliver the result of `JSON.stringify` into `response.end`.

Back on the client side, our third argument in the first `$.get` call is '`json`', this ensures `$.get` interprets the incoming data as JSON, converting it to an object. The object is passed in as the first parameter of the function callback of `$.get` (second parameter of `$.get`), which we named `profile_names`. We use jQuery's `$.each` to loop through the `profile_names`, populating the first `select` element (`#profiles`) by applying jQuery's `append` method to the element, and adding each profile name inside the `<option>` elements as we loop through `$.each`.

Next, we apply a listener to our two `select` elements (`change`) whose callback assembles a URL dependent upon the user's selection, passing this URL into another AJAX request using `$.get`.

This time on the server side, the `/profile route` method is invoked, passing in the corresponding profile from our `profiles` object to `output`. This property will contain an object holding the profile information of the requested individual.

In our second `$.get` call, we set the third argument to '`text`'. This will force jQuery not to automatically interpret incoming data as JSON or XML. This gives us more control and makes it easier to output the raw data into `textarea`. Inside the `$.get` callback, we use the `jqXHR` parameter to determine the `Content-Type` to see if we have JSON or XML. We loop through the returned data according to its type (either `Object` or `XMLObject`) and append it to our `#output div`.

There's more...

We can also convert our objects to JSON and XML in the browser and send them over to our server, where we can interact with them as objects again.

Sending serialized data from client to server

Let's extend our example to add new profiles to our `profiles` object on the server using our browser interface.

Starting with `index.html` (which we'll copy to `add_profile_index.html`—we'll also copy `server.js` to `add_profile_server.js`), let's append a form called `#add`, and style it. Here's the form:

```
<form id=add>
<div><label>profile name</label><input name="profileName"></div>
<div><label>name</label><input name="name"></div>
<div><label>irc</label><input name="irc"></div>
<div><label>twitter</label><input name="twitter"></div>
<div><label>github</label><input name="github"></div>
<div><label>location</label><input name="location"></div>
<div><label>description</label><input name="description"></div>
<div><button>Add</button></div>
</form>
```

And some additional styles:

```
<style>
#frm, #raw {display: block; float: left; width: 210px}
#raw {height: 150px; width: 310px; margin-left: 0.5em}
#add {display: block; float: left; margin-left: 1.5em}
#add div {display: table-row}
#add label {float: left; width: 5.5em}
div button {float: right}
</style>
```

We're going to be using our `buildXml` function on the client side (we created `buildXml` in the *Converting an object to XML and back again* recipe). This function is already available on our server, so we'll make it available to the client by converting it to a string on server starts and supplying a route for the client to access it:

```
var index = fs.readFileSync('add_profile_index.html');
var buildXmljs = buildXml.toString();
var routes,
  mimes = {
  js: "application/javascript",
```

```

    json: "application/json",
    xml: "application/xml"
};

routes = {
  'profiles': function(format) {
    return output(Object.keys(profiles), format);
  },
  '/profile': function(format, basename) {
    return output(profiles[basename], format, basename);
  },
  'buildXml' : function(ext) {
    if (ext === 'js') { return buildXmljs; }
  }
};

```

We also updated our `names` object ready to deliver `application/javascript` Content-Type and altered the `index` variable to use our new `add_profile_index.html` file. Back in our client-side code, we fetch our `buildXml` function by including another `<script>` tag in the head section:

```
<script src=buildXml.js></script>
```

We'll wrap our initial `$.get` call to the server (which fetches all the profile names for the `select` element) in a function called `load`. This allows us to dynamically reload the profile names once a profile has been added:

```

function load() {
  $.get('http://localhost:8080/profiles',
    function(profile_names) {
      $.each(profile_names, function(i, pname) {
        $('#profiles').append('<option>' + pname + '</option>');
      });
    },
    'json'
  );
}

load();

```

Now we define a handler for the `#add` form:

```

$('#add').submit(function(e) {
  var output, obj = {}, format = $('#formats').val();
  e.preventDefault();
  $.each($(this).serializeArray(), function(i, nameValuePair) {
    obj[nameValuePair.name] = nameValuePair.value; // form an object
  });
}

```

```
    output = (format === 'json') ? JSON.stringify(obj) :
buildXml(obj, 'xml');

$.ajax({ type: 'POST', url: '/', data: output,
  contentType: 'application/' + format, dataType:
'text',
  success: function(response) {
    $('#raw').val(response);
    $('#profiles').html('<option></option>');
    load();
  }
});
});
```

Our handler builds an object from the form input serializing it to the specified format. It uses `jQuery.ajax` to send serialized data to our server, and afterwards reloads the profiles. On our server, we'll write a function to handle the POST request:

```
function addProfile(request, cb) {
  var newProf, profileName, pD = '' // post data
  request
    .on('data', function(chunk) { pD += chunk; })
    .on('end', function() {
      var contentType = request.headers['content-type'];
      if (contentType === 'application/json') {
        newProf = JSON.parse(pD);
      }

      if (contentType === 'application/xml') {
        xm2js.parseString(pD, function(err, obj) {
          newProf = obj;
        });
      }
      profileName = newProf.profile.name;
      profiles[profileName] = newProf;
      delete profiles[profileName].profile.name;
      cb(output(profiles[profileName],
        contentType.replace('application/', '') ,
        profileName));
    });
}
```

For our new `addProfile` function to work, we need to include the `xml2js` module which is used to convert the serialized XML back into an object. So alongside all our initial variables we add the following:

```
var xml2js = new (require('xml2js')).Parser();
```

As in the first recipe of *Chapter 2, Exploring the HTTP Object*, in handling POST data, `addProfile` compiles all the incoming data together. In the end event, we convert the serialized data to an object using the method appropriate to its type. We take this object and add it to our `profiles` object using the `profileName` property as the key for our sub-object. Once we've added the object, we delete the redundant `profileName` property.

To return data to the client, the `addProfile` function invokes the callback (`cb`) parameter, passing in our custom `output` function which will return serialized data according to the specified format (which is determined by using `replace` on the `Content-Type` header).

We include our `addProfile` function in our server like so:

```
http.createServer(function (request, response) {
  //initial server variables...
  if (request.method === 'POST') {
    addProfile(request, function(output) {
      response.end(output);
    });
    return;
  }
  //...rest of the server code (GET handling...)
});
```

Within our `addProfile` callback function, we simply end the response with the data returned from the `output` function, accessing this data via the `output` parameter as defined in the `addProfile` callback. The new profiles are only saved in operational memory, so they will be lost on server restarts. If we were to store this data on disc, we would ideally want to save it in a database, which we'll talk about in the next chapter, *Interfacing with Databases*.

See also

- ▶ *Setting up a router* discussed in *Chapter 1, Making a Web Server*
- ▶ *Processing POST data* discussed in *Chapter 2, Exploring the HTTP Object*
- ▶ *Converting an object to JSON and back again* discussed in this chapter
- ▶ *Converting an object to XML and back again* discussed in this chapter

Working with real data: fetching trending tweets

Many online entities format their response data as JSON and XML in their Application Programmer Interfaces (APIs) to expose pertinent information to third-party developers who can subsequently integrate this data into their applications.

One such online entity is Twitter. In this recipe, we are going to make a command-line application that makes two requests to Twitter's REST service. The first will retrieve the most popular current topics on Twitter and the second will return the most recent tweets regarding the hottest topic on Twitter.

Getting ready

Let's create our file and name it `twitter_trends.js`. We may also wish to install the third-party `colors` module to make our output more beautiful:

```
npm install colors
```

How to do it...

We'll need the `http` module in order to make requests, and the `colors` module to get some color in our console output:

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

We're going to be making a GET request inside another GET request. Between these requests we'll be processing JSON data to either pass into the subsequent request or to output to console. In the spirit of DRY (Don't Repeat Yourself), and to demonstrate how to avoid spaghetti code, we'll abstract our GET requests and JSON handling into a function called `makeCall`.

```
function makeCall(urlOpts, cb) {
  http.get(urlOpts, function(response) { // make a call to the twitter API
    trendingTopics.jsonHandler(response, cb);
  }).on('error', function(e) {
    console.log("Connection Error: " + e.message);
  });
}
```

Notice the mysterious appearance of `trendingTopics` and its `jsonHandler` method. `trendingTopics` is an object that is going to provide all the settings and methods for our Twitter interactions. `jsonHandler` is a method on the `trendingTopics` object for receiving the response stream and converting the JSON to an object.

We need to set up options for our calls to the trends and tweets APIs, along with some Twitter interaction-related functionality. So above our `makeCall` function, we'll create the `trendingTopics` object as follows:

```
var trendingTopics = module.exports = {
  trends: {
    urlOpts: {
      host: 'api.twitter.com',
      path: '/1/trends/1.json', // 1.json provides global trends,
      headers: {'User-Agent': 'Node Cookbook: Twitter Trends'}
    }
  },
  tweets: {
    maxResults: 3, // twitter applies this very loosely for the "mixed"
    type: 'mixed',
    resultsType: 'realtime', // choice of mixed, popular or realtime
    language: 'en', // ISO 639-1 code
    urlOpts: {
      host: 'search.twitter.com',
      headers: {'User-Agent': 'Node Cookbook: Twitter Trends'}
    }
  },
  jsonHandler: function (response, cb) {
    var json = '';
    response.setEncoding('utf8');
    if (response.statusCode === 200) {
      response.on('data', function (chunk) {
        json += chunk;
      }).on('end', function () {
        cb(JSON.parse(json));
      });
    } else {
      throw ("Server Returned statusCode error: " + response.statusCode);
    }
  },
  tweetPath: function (q) {
    var p = '/search.json?lang=' + this.tweets.language + '&q=' + q +
      '&pp=' + this.tweets.maxResults + '&include_entities=true' +
      '&with_twitter_user_id=true&result_type=' +
      this.tweets.resultsType;
    this.tweets.urlOpts.path = p;
  }
};
```

While creating the `trendingTopics` variable, we also turn the object into a module by simultaneously loading it into `module.exports`. See how we use this in the *There's more...* section.

Within our `trendingTopics` object, we have the `trends` and `tweets` objects and two methods: `jsonHandler` and `tweetPath`.

Finally, we'll invoke our `makeCall` function to request the top global trends from the Twitter trends API, converting the returned JSON to an object, AND using this object to ascertain the path to request tweets on the highest trending topic for using another embedded `makeCall` invocation.

```
makeCall(trendingTopics.trends.urlOpts, function (trendsArr) {
  trendingTopics.tweetPath(trendsArr[0].trends[0].query);
  makeCall(trendingTopics.tweets.urlOpts, function (tweetsObj) {
    tweetsObj.results.forEach(function (tweet) {
      console.log("\n" + tweet.from_user.yellow.bold + ': ' + tweet.text);
    });
  });
});
```

How it works...

Let's pick apart the `trendingTopics` object. `trends` and `tweets` provide settings relevant to Twitter's API. For `trends` this is simply a URL options object to be passed into `http.get` later on. In the `tweets` object, we have the URL object along with some other properties pertaining to options we can set within our REST call to the Twitter search API.

Twitter API and the User-Agent header

Notice we've gone to the trouble of setting a `User-Agent` header. This is due to the Twitter API policy, which penalizes a lack of `User-Agent` string by imposing a lower rate limit.

Our `jsonHandler` method on the `trendingTopics` object takes a `response` and `cb` (callback) parameter. `trendingTopics.jsonHandler` uses the `response` object from the `http.get` call to capture the incoming data stream into a variable (`json`). When the stream has ended, which is detected using the `end` event listener on `response`, `cb` is invoking converted JSON as the parameter. The callback from `trendingTopics.jsonHandler` finds its way up into the `makeCall` callback.

`makeCall` abstractly combines the GET request and JSON handling, and provides a callback with a single parameter which is the data returned by Twitter as parsed JSON (in this case, it is an array of objects).

In the outer `makeCall` invocation we call the parameter `trendsArr`, because Twitter returns its JSON data in an array wrapper. We use `trendsArr` to locate the query fragment representation of the Twitter's top trend and pass it to the final method of our `trendingTopics` object: `trendingTopics.tweetPath`. This method takes a query fragment (`q`) as its single parameter. It then uses this parameter along with the options in `trendingTopics.tweets` to build the final Search API path. It injects this path into the `urlOpts` object of `trendingTopics.tweets`, which is then passed through into the inner `makeCall` invocation.

In the inner `makeCall` invocation we name the parameter `tweetsArr`. This is an array of objects containing tweet data as returned from the Twitter Search API in response to a query searching for the top trend discovered via the former (outer) call to the Trend API. We loop through the array using the veritable `for Each` (ES5) looping function, handling each element passed through the loop as `tweet`.

The objects contained in the `tweetsArr` array contain lots of data, such as time information, amount of re-tweets, and so forth. However, we're just interested in the content of the tweet, and who tweeted. So we log the `from_user` and `text` properties of each `tweet` to the console:

```
node@cookbook~$ node twitter_trends.js
LemonMonster: #100thingsilove 49. truffles
KAl德erson1D: #100thingsilove 22) Liam Payne
AnaCalderonx7: #100thingsilove 28- Comidaaaaaaa :D
node@cookbook~$
```

This is also where the `colors` module comes in handy since, within `console.log` we have `tweet.from_user.yellow.bold`. The colors are not properties on the object returned by Twitter, but rather some trickery performed by the `colors` module to provide an easy interface for styling console text.

There's more...

Let's look into working with an XML-based service.

Cross referencing Google Hot Trends with Twitter tweets

It may be noticed that trending tweets tend to have rather fad-like influences generated from within the Twitter community. Google Hot Trends is another source of trending information. It provides hourly updates of the highest trending searches.

We can extend our example to access and process Google's Hot Trends XML atom feed, and then integrate the top result into our Twitter Search API request. To do this, let's create a new file called `google_trends.twitter.js`. It's nice to work with XML data as a JavaScript object, so we'll require the non-core `xml2js` featured in the *Converting an object to XML and back again* recipe in this chapter, along with `http`, `colors`, and our own `trendingTopics` module.

```
var http = require('http');
var xml2js = new (require('xml2js')).Parser();
var colors = require('colors'); //for prettifying the console output
var trendingTopics = require('./twitter_trends'); //load
trendingTopics.obj
```

Now we'll extend our `trendingTopics` object by inheriting from it using the EcmaScript 5 `Object.create` method.

```
var hotTrends = Object.create(trendingTopics, {trends: {value:
urlOpts: {
host: 'www.google.com',
path: '/trends/hot-trends/atom/hourly',
headers: {'User-Agent': 'Node Cookbook: Twitter Trends'}
}
}});

hotTrends.xmlHandler = function(response, cb) {
  var hotTrendsfeed = '';
  response.on('data', function(chunk) {
    hotTrendsfeed += chunk;
  }).on('end', function() {
    xml2js.parseString(hotTrendsfeed, function(err, obj) {
      if (err) { throw (err.message); }
      xml2js.parseString(obj.entry.content['#'],
        function(err, obj) {
          if (err) { throw (err.message); }
          cb(decodeURIComponent(obj.li[0].span.a['#']));
        });
    });
  });
};
```

We declared a variable called `hotTrends`, and used `Object.create` to initialize an instance of `trendingTopics`, re-substantiating the `trends` property via the property declarations object (the second parameter of `Object.create`). This means that instead of `trends` being an inherited property, it now belongs to `hotTrends` and we haven't overwritten the `trends` property in `trendingTopics` when adding it to our new `hotTrends` object.

We then add a new method: `hotTrends.xmlHandler`. This combines all the incoming chunks into the `hotTrends.feed` variable. Once the stream has ended, it invokes `xml2js.parseString` and passes the XML contained in `hotTrends.feed` into it. In the callback of the first `parseString` method, we invoke `xml2js.parseString` again. Why? Because we have to parse two sets of XML, or rather one set of XML and one set of adequately formed HTML. (If we head to `http://www.google.com/trends/hotrends/atom/hourly` it will be rendered as HTML. If we view the source, we'll then see an XML document with embedded HTML content.)

Google's Hot Trends XML feed delivers the Hot Trends as HTML inside of its `content` XML node.

The HTML is wrapped within a CDATA section, so it isn't parsed by `xml2js` the first time round. Ergo, we create a new `Parser` and then parse the HTML via `obj.entry.content['#']`.

Finally, the `hotTrends.xmlHandler` method completes in the second embedded `xml2js` callback where it executes its own callback parameter (`cb`) with a query fragment generated from the top list item element in HTML.

Now all we have to do is make some adjustments to `makeCall`:

```
function makeCall(urlOpts, handler, cb) {
    http.get(urlOpts, function(response) { // make a call to the twitter api
        handler(response, cb);
    }).on('error', function(e) {
        console.log("Connection Error: " + e.message);
    });
}

makeCall(hotTrends.trends.urlOpts, hotTrends.xmlHandler, function(query) {
    hotTrends.tweetPath(query);
    makeCall(hotTrends.tweets.urlOpts, hotTrends.jsonHandler, function(tweetsObj) {
        tweetsObj.results.forEach(function(tweet) {
            console.log("\n" + tweet.from_user.yellow.bold + ': ' + tweet.text);
        });
    });
});
```

As we are now dealing with both JSON and XML, we slipped in another parameter to our `makeCall` function declaration: `handler`. The `handler` parameter allows us to specify whether to use the inherited `jsonHandler` method or our supplemented `xmlHandler` method.

When we invoke the outer `makeCall`, we pass in `hotTrends.xmlHandler`, naming the parameter `query`. This is done because we are directly passing in the query fragment generated by `xmlHandler` instead of the array returned from Twitter. This is passed directly into the `tweetPath` method, which consequently updates the `path` property of the `hotTrends.tweets.urlOpts` object.

We pass `hotTrends.tweets.urlOpts` into the second `makeCall`, this time setting the `handler` parameter to `hotTrends.jsonHandler`.

The second `makeCall` callback behaves exactly the same as in the main recipe. It outputs the tweets to the console. This time, however, it outputs tweets based on Google Hot Trends.

See also

- ▶ *Using Node as an HTTP client* discussed in *Chapter 2, Exploring the HTTP Object*
- ▶ *Converting an object to JSON and back again* discussed in this chapter
- ▶ *Converting an object to XML and back again* discussed in this chapter

4

Interfacing with Databases

In this chapter we will cover:

- ▶ Writing to a CSV file
- ▶ Connecting and sending SQL to a MySQL server
- ▶ Storing and retrieving data with MongoDB
- ▶ Storing and retrieving data with Mongoskin
- ▶ Storing data to CouchDB with Cradle
- ▶ Retrieving data from CouchDB with Cradle
- ▶ Accessing CouchDB changes stream with Cradle
- ▶ Storing and retrieving data with Redis
- ▶ Implementing PubSub with Redis

Introduction

As the complexity of our code and the demands of our objectives increase, we soon realize the need for a place to store our data.

We then have to ask the question: What is the best way to store our data? The answer depends on the type of data we are working with since different challenges require different solutions.

If we're doing something very simple, we could save our data as a flat CSV file, which has the added benefit of enabling users to view the CSV file in a spreadsheet application.

If we are working with data that has clearly relational qualities, for instance accounting data whereby there are clear, distinct relationships between two sides of a transaction, then we would choose a relational database such as the popular MySQL.

In many cases relational databases became a de facto standard for nearly all data scenarios. This led to the necessity of imposing relationships upon otherwise loosely-related data (such as website content) in an attempt to squeeze it into our relational mental model.

In recent times, though, there has been a movement away from relational databases towards NoSQL, a non-relational paradigm. The driving force being that we tailor our technology to best suit our data rather than trying to fit our data into our technology.

In this chapter, we will look into various data storage technologies with examples of their usage in Node.

Writing to a CSV file

A flat file structure is one of the most elementary database models. Columns can either be fixed length, or use delimiters. The Comma Separated Values (CSV) convention conforms to the idea of delimited flat file structure databases. While it's called CSV, the term CSV is also applied as a broad blanket term for any basic delimited structure consisting of one record per line (for example, tab-separated values).

We could follow a brittle approach for constructing CSV structures, simply by using a multidimensional array and the `join` method:

```
var data = [['a','b','c','d','e','f','g'],
            ['h','i','j','k','l','m','n']];
var csv = data.join("\r\n"); /* renders: a,b,c,d,e,f,g
                            h,i,j,k,l,m,n */
```

However, the limitations of this technique quickly become apparent. What if one of our fields contains a comma? Now one field becomes two, thus corrupting our data. Furthermore, we are limited to just using commas as delimiters.

In this recipe we will use the third-party `ya-csv` module to store data in the CSV format.

Getting ready

Let's create a file called `write_to_csv.js`, we'll also need to retrieve `ya-csv`.

```
npm install ya-csv
```

How to do it...

We require the `ya-csv` module, call its `createCsvFileWriter` method to instantiate a CSV file writer and loop through our array, calling the `writeRecord` method of the CSV file writer:

```
var csv = require('ya-csv');
var writer = csv.createCsvFileWriter('data.csv');

var data = [[ 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' ], [ 'h' , 'i' , 'j' , 'k' , 'l' , 'm' ,
'n' ]];

data.forEach(function(rec) {
  writer.writeRecord(rec);
});
```

Let's take a look at the file we saved to, `data.csv`:

```
"a", "b", "c", "d", "e", "f", "g"
"h", "i", "j", "k", "l", "m", "n"
```

How it works...

The difficulty with writing to and reading from CSV files is the edge cases, such as commas or quotes embedded in text. `ya-csv` handles these edge cases for us.

We load an instance of `CsvWriter` of `ya-csv` into the `writer` variable with `createCsvFileWriter`.

Then we simply loop through each array as `rec`, passing it to the `writeRecord` method of `CsvWriter` of `ya-csv`. Behind the scenes it re-factors each array and passes it to an instance of `fs.WriteStream`.

This recipe depends upon us using a basic data structure in our code. Multidimensional objects would have to be massaged into the right format as `writeRecord` only functions with arrays.

There's more...

Could we easily create this functionality ourselves? Without a doubt. However, `ya-csv` provides an API for us to seamlessly customize the elements of our CSV files, and implements the more involved CSV parsing functionality.

Customizing the CSV elements

If we save our recipe file as `write_to_custom_csv.js`, and pass an `options` object to `createCsvFileWriter`, we can alter the way our CSV files are constructed:

```
var writer = csv.createCsvFileWriter('custom_data.csv', {
  'separator': '~',
  'quote': '|',
  'escape': '|'
});
```

Notice the `escape` option. This sets the character which will prevent accidental closure of a CSV field. Let's slip one into our array to see how `ya-csv` handles it:

```
var data = [['a','b','c','d','e||','f','g'],
['h','i','j','k','l','m','n']];
```

After running our new code, let's take a look at `custom_data.csv`:

```
| a|~| b|~| c|~| d|~| e|||~| f|~| g|
| h|~| i|~| j|~| k|~| l|~| m|~| n|
```

See how our pipe character in the `e` field has another pipe appended to it in order to escape it.

Reading a CSV file

We can also use `ya-csv` to read from a CSV file, its built-in parser converts each CSV record back into an array. Let's make `read_from_csv.js`.

```
var csv = require('ya-csv');
var reader = csv.createCsvFileReader('data.csv');
var data = [];

reader.on('data', function(rec) {
  data.push(rec);
}).on('end', function() {
  console.log(data);
});
```

If we want it to parse alternative delimiters and quotes, we simply pass these into the `options` object of `createCsvFileReader`:

```
var reader = csv.createCsvFileReader('custom_data.csv', {
  'separator': '~',
  'quote': '|',
  'escape': '|'
});
```

Manipulating CSV as a stream

`ya-csv` interacts with CSV files as streams. This can reduce operational memory since streams allow us to process small chunks of information as they are loaded, instead of buffering the entire file into memory first.

```
var csv = require('ya-csv');
var http = require('http');

http.createServer(function (request, response) {
  response.write('[');
  csv.createCsvFileReader('big_data.csv')
    .on('data', function(data) {
      response.write(((this.parsingStatus.rows > 0) ? ',' : '') + JSON.stringify(data));
    })
    .on('end', function() {
      response.end(']');
    });
}).listen(8080);
```

See also

- ▶ *Connecting and sending SQL to a MySQL server* discussed in this chapter
- ▶ *Storing and retrieving data with Mongoskin* discussed in this chapter
- ▶ *Storing data to CouchDB with Cradle* discussed in this chapter
- ▶ *Storing and retrieving data with Redis* discussed in this chapter

Connecting and sending SQL to a MySQL server

Structured Query Language has been a standard since 1986 and it's the prevailing language for relational databases. MySQL is the most popular SQL relational database server around, often appearing in the prevalent LAMP (Linux Apache MySQL PHP) stack.

If a relational database was conceptually relevant to our goals in a new project, or we were migrating a MySQL-backed project from another framework to Node, the third-party `mysql` module would be particularly useful.

In this task, we will discover how to connect to a MySQL server with Node and execute SQL queries across the wire.

Getting ready

Let's grab `mysql`, which is a pure JavaScript (as opposed to C++ bound) MySQL client module.

```
npm install mysql
```

We'll need a MySQL server to connect to. By default, the `mysql` client module connects to `localhost`, so we'll have MySQL running locally.

On Linux and Mac OSX we can see if MySQL is already installed with the following command:

```
whereis mysql
```

And we can see if it is running using the following command:

```
ps -ef | grep mysqld
```

If it is installed but not running, we can do:

```
sudo service mysql start
```

If MySQL isn't installed we can use the relevant package manager for our system (homebrew, apt-get/synaptic, yum, and so on), or if we're using Node on Windows we can head to <http://dev.mysql.com/downloads/mysql> and download the installer.

Once we're ready to go, let's create a file and call it `mysql.js`.

How to do it...

First, we require the third-party `mysql` driver, and create a connection to the server:

```
var mysql = require('mysql');
var client = mysql.createClient({
  user: 'root',
  password: 'OURPASSWORD',
  // debug: true
});
```

We need a database to connect to. Let's keep things interesting and make a `quotes` database. We can do that by passing SQL to the `query` method:

```
client.query('CREATE DATABASE quotes');
client.useDatabase('quotes');
```

We've also called the `useDatabase` method to connect to the database, although we could achieve the same with `client.query('USE quotes')`. Now we'll create a table by the same name.

```
client.query('CREATE TABLE quotes.quotes (' +
    'id INT NOT NULL AUTO_INCREMENT, ' +
    'author VARCHAR(128) NOT NULL, ' +
    'quote TEXT NOT NULL, PRIMARY KEY (id)' +
    ')');
```

If we were to run our code more than once, we'd notice that an unhandled error is thrown and the program fails. This is due to the `mysql` driver emitting an error event in reflection of a MySQL server error. It's throwing an unhandled error because the `quotes` database (and table) cannot be created as they already exist.

We want our code to be versatile enough to create a database if necessary, but not throw an error if it's not there. To do this, we're going to catch any errors emitted by our client instance, filtering out the database/table exists errors:

```
var ignore = [mysql.ERROR_DB_CREATE_EXISTS,
    mysql.ERROR_TABLE_EXISTS_ERROR];

client.on('error', function(err) {
    if(ignore.indexOf(err.number) > -1) { return; }
    throw err;
});
```

We'll place our error catcher just before the `client.query` method invocations. Finally, at the end of our code, we'll insert our first quote into the table and send a `COM_QUIT` packet (using `client.end`) to the MySQL server. This will only close the connection once all queued SQL has been executed.

```
client.query('INSERT INTO quotes.quotes (' +
    'author, quote) ' +
    'VALUES ("Bjarne Stroustrup", "Proof by analogy is
fraud.");');

client.end();
```

How it works...

The `createClient` method establishes a connection to the server and returns a client instance for us to interact with. We can pass it as an `options` object that may contain `host`, `port`, `user`, `password`, `database`, `flags`, and `debug`. Other than `user` and `password`, the default options are fine for our purposes though. If we uncomment `debug` we can see the raw data being sent to and from the server.

`client.query` sends SQL to our database which is then executed by the MySQL server. With it, we CREATE a DATABASE named `quotes` and also a TABLE named `quotes`. Then we INSERT our first record (a quote by the inventor of C++) into our database.

`client.query` queues each piece of SQL passed to it, executing statements asynchronously alongside our other code, but sequentially within the SQL statement queue. When we call `client.end`, the connection closing task is added to the end of the queue. If we wanted to disregard the statement queue, and immediately end the connection, we would use `client.destroy`.

Our `ignore` array holds two numbers, 1007 and 1050—we grab these numbers from the `mysql` object, which holds MySQL error codes. We want to ignore MySQL errors that occur when a table or database already exists, otherwise we could only run `mysql.js` once. After the first run it would crash since the database and table would already exist. Ignoring these codes means we can implicitly set up our database, and have just one file rather than a setup app and a separate app for inserting code.

In the `error` event listener we check if `err.number` is within our `ignore` array. If it is, we simply `return`, thus ignoring the error and gracefully continuing execution. If the error is of some other nature, we fall through to the usual behavior of throwing the error.

There's more...

We don't just send data to MySQL, we retrieve it. Furthermore, SQL queries are often generated from user input, but this can be open to exploitation if precautions aren't taken.

Using and cleaning user input

As in other languages that build SQL statements with string concatenation, we must prevent the possibilities of SQL injection attacks to keep our server safe. Essentially, we must clean (which is, escape) any user input to eradicate the potential for unwanted SQL manipulation.

We'll copy `mysql.js` and call it `insert_quotes.js`. To implement the concept of user input in a simple way we'll pull arguments from the command line, but the principles and methods of data cleaning extend to any input method (for example, via a query string on a request).

Our basic API will go like this:

```
node quotes.js "Author Name" "Quote Text Here"
```

Quotation marks are essential to divide the command-line arguments, but for the sake of brevity we won't be implementing any validation checks.



Command-line parsing module: optimist

For more advanced command-line functionality, check out the excellent `optimist` module, available at <https://www.github.com/substack/node-optimist>.

To receive an author and quote, we'll load the two quotes arguments into a new `params` object.

```
var params = {author: process.argv[2], quote: process.argv[3]};
```

Our first argument is at 2 in the `process.argv` array because 0 and 1 hold `node` and `quotes.js`.

Now let's slightly modify our `INSERT` statement:

```
if (params.author && params.quote) {
  client.query('INSERT INTO quotes.quotes (' +
    'author, quote) ' +
    'VALUES (?, ?)', [params.author, params.quote]);
}
client.end();
```

We've placed this just before our final `client.end` call in the main recipe. The `mysql` module can seamlessly clean user input for us. We simply use the question mark (?) as a placeholder and then pass our values (in order) as an array into the second parameter of `client.query`.

Receiving results from the MySQL server

Let's extend `insert_quotes.js` further by outputting all quotes for an author, whether or not a quote is provided. We'll save `insert_quotes.js` simply as `quotes.js`.

Underneath our `INSERT` query, but above the final `client.end`, we'll add the following code:

```
if (params.author) {
  client.query('SELECT * FROM quotes WHERE ' +
    'author LIKE ' + client.escape(params.author))
  .on('row', function (rec) {
    console.log('%s: %s\n', rec.author, rec.quote);
  });
}
client.end();
```

On this occasion, we've used an alternative approach to clean user input with `client.escape`. This has exactly the same effect as the former, but only escapes a single input. Generally, if there's more than one variable, the former method would be preferable.

The results of a SELECT statement can be accessed either by passing a callback function, or by listening for the `row` event. A `row` event listener allows us to interact with a MySQL server data stream, one row at a time.

We can safely call `client.end` without putting it in the `end` event of our SELECT query because `client.end` only terminates a connection when all queries are done.

See also

- ▶ *Storing and retrieving data with MongoDB* discussed in this chapter
- ▶ *Storing and retrieving data with Redis* discussed in this chapter

Storing and retrieving data with MongoDB

MongoDB is a NoSQL database offering that maintains a philosophy of performance over features. It's designed for speed and scalability. Instead of working relationally, it implements a document-based model that has no need for schemas (column definitions). The document model works well for scenarios where relationships between data are flexible and where minimal potential data loss is an acceptable cost for the speed enhancements (a blog for instance).

While it's in the NoSQL family, MongoDB attempts to sit between two worlds, providing a syntax reminiscent of SQL but operating non-relationally.

In this task, we'll implement the same quotes database as in the previous recipe, using MongoDB instead of MySQL.

Getting ready

We'll want to run a MongoDB server locally. It can be downloaded from <http://www.mongodb.org/downloads>.

Let's start the MongoDB service, `mongod`, in the default debug mode:

```
mongod --dbpath [a folder for the database]
```

This allows us to observe the activities `mongod` as it interacts with our code, if we wanted to start it as a persistent background service we would use.

```
mongod --fork --logpath [p] --logappend --dbpath [p]
```

Where `[p]` is our desired path.



More information on starting and correctly stopping `mongod` can be found at <http://www.mongodb.org/display/DOCS/Starting+and+Stopping+Mongo>.

To interact with MongoDB from Node, we'll need to install the `mongodb` native bindings driver module.

```
npm install mongodb
```

We'll also create a new folder for our MongoDB-based project, with a new `quotes.js` file.

How to do it...

We must require the `mongodb` driver, initiate a MongoDB server instance, and create a client which loads the `quotes` database and connects to the MongoDB server.

```
var mongo = require('mongodb');
var server = new mongo.Server("localhost", 27017);
var client = new mongo.Db('quotes', server);

var params = {author: process.argv[2], quote: process.argv[3]};
```

Notice we've also inserted our `params` object for reading user input from the command line.

Now we open a connection to our `quotes` database and load (or create if necessary) our `quotes` collection (a table would be the closest similar concept in SQL).

```
client.open(function (err, client) {
  if (err) { throw err; }
  var collection = new mongo.Collection(client, 'quotes');
  client.close();
});
```

Next, we'll insert a new document (in SQL terms this would be a record) according to a user-defined author and quote.

We'll also output any quotes by the specified author to the console.

```
client.open(function (err, client) {
  if (err) { throw err; }

  var collection = new mongo.Collection(client, 'quotes');

  if (params.author && params.quote) {
    collection.insert({author: params.author, quote: params.quote});
  }

  if (params.author) {

    collection.find({author: params.author}).each(function (err, doc)
{
```

```
        if (err) { throw err; }
        if (doc) { console.log('%s: %s \n', doc.author, doc.quote);
      return;
    client.close();
  });

return;
}

client.close();
});
```

We can see our MongoDB backed quotes application in action in the following screenshot:

```
Node Cookbook
File Edit View Search Terminal Help
node@cookbook:~$ node quotes.js "Albert Einstein" "Imagination is more important than knowledge"
Albert Einstein: Imagination is more important than knowledge

node@cookbook:~$ node quotes.js "Ronald Reagan" "You can tell a lot about a fellow's characters by his way of eating jellybeans"
Ronald Reagan: You can tell alot about a fellow's characters by his way of eating jellybeans

node@cookbook:~$ node quotes.js "Albert Einstein" "It's not that I'm so smart, it's just that I stay with problems longer"
Albert Einstein: Imagination is more important than knowledge

Albert Einstein: It's not that I'm so smart, it's just that I stay with problems longer

node@cookbook:~$
```

How it works...

When we create a new instance of `mongo.Db`, we pass in the name of the database as the first parameter. MongoDB will intelligently create this database if it doesn't exist.

We use the `open` method of the `Db` instance, which we named `client`, to open a connection to our database. Once the connection is made, our callback function is executed where we can interact with the database via the `client` parameter.

We start off by making a `Collection` instance. `Collection` is similar to an SQL table in that it holds all our database fields. However, rather than field values being grouped by column, a collection contains multiple documents (like records) where each field holds both the field name and its value (documents are very much like JavaScript objects).

If both `quote` and `author` are defined, we invoke the `insert` method of our `Collection` instance, passing in an object as our document.

Finally, we use `find` which is comparable to the SELECT SQL command, passing in an object that specifies the author field and desired value. The `mongodb` driver provides a convenience method (`each`) that can be chained with the `find` method. `each` executes the callback passed to it for each document as it's found. The last loop of `each` passes in `doc` as `null`, which conveniently signals that MongoDB has returned all records.

So as long as `doc` is truthy, we pass the `author` and `quote` properties of every `doc` found. Once `doc` is `null`, we allow the interpreter to discover the last part of the callback, `client.close`, by not returning early from the callback.

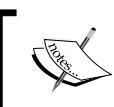
The second and final `client.close` situated at the end of the `client.open` callback is invoked only when there are no arguments defined via the command line.

There's more...

Let's check out some other useful MongoDB features.

Indexing and aggregation

Indexing causes MongoDB to create a list of values from a chosen field. Indexed fields accelerate query speeds because a smaller set of data can be used to cross reference and pull from a larger set. We can apply an index to the author field and see performance benefits, especially as our data grows. Additionally, MongoDB has various commands allowing us to aggregate our data. We can group, count, and return distinct values.



For more advanced needs or larger sets of data, map/reduce functions can aggregate. CouchDB also uses map/reduce to generate views (stored queries). See *Retrieving data from CouchDB with Cradle*.



Let's create and output a list of authors found in our database, and save our code to a file called `authors.js`.

```
var mongo = require('mongodb');
var server = new mongo.Server("localhost", 27017);
var client = new mongo.Db('quotes', server);

client.open(function (err, client) {
  if (err) { throw err; }
  var collection = new mongo.Collection(client, 'quotes');
```

```
collection.ensureIndex('author', {safe: true}, function (err) {
  if (err) { throw err; }
  collection.distinct('author', function (err, result) {
    if (err) { throw err; }
    console.log(result.join('\n'));
    client.close();
  });
});
```

As usual, we opened up a connection to our `quotes` database, grabbing our `quotes` collection. Using `ensureIndex` creates an index only if one doesn't already exist. We pass in `safe: true` so that MongoDB returns any errors and our callback works as expected. Inside the callback, we invoke the `distinct` method on our `collection`, passing in `author`. The result is passed as an array through `result`. We `join` the array into a string using new lines and output to the console. .

Updating modifiers, sort, and limit

We could make it possible for a hypothetical user to indicate if they were inspired by a quote (such as a **Like** button), then we could use the `sort` and `limit` commands to output the top ten most inspiring quotes.

In reality, this would be implemented with some kind of user interface (for example, in a browser), but we'll again emulate user interactions using the command line; let's create a new file called `quotes_votes.js`.

First, in order to vote for a quote, we'll need to reference it, which can be done by the unique `_id` property. So in `quotes_votes.js` let's write:

```
var mongo = require('mongodb');
var server = new mongo.Server("localhost", 27017);
var client = new mongo.Db('quotes', server);
var params = {id: process.argv[2], voter: process.argv[3]};
client.open(function (err, client) {
  if (err) { throw err; }
  var collection = new mongo.Collection(client, 'quotes');

  // vote handling to go here

  collection.find().each(function (err, doc) {
    if (err) { throw err; }
    if (doc) { console.log(doc._id, doc.quote); return; }
    client.close();
  });
});
```

Now when we run `quotes_votes.js` with node, we'll see a list of IDs and quotes. To vote for a quote, we'll simply copy an ID and use it as our command-line parameter. So let's do our vote handling as shown in the following code:

```

if (params.id) {
  collection.update({_id: new mongo.ObjectID(params.id)},
    {$inc: {votes: 1}}, {safe: true}
    function (err) {
      if (err) { throw err; }

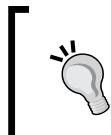
      collection.find().sort({votes: -1}).limit(10).each(function
        (err, doc) {
          if (err) { throw err; }
          if (doc) {
            var votes = (doc.votes) || 0;
            console.log(doc.author, doc.quote, votes);
            return;
          }
          client.close();
        });
    });
  return;
}

```

MongoDB IDs must be encoded as a BSON (Binary JSON) ObjectID. Otherwise, the `update` command will look for `param id` as a string, failing to find it. So we create a new `mongo.ObjectID(param id)` to convert `param id` from a JavaScript string to a BSON ObjectId.

`$inc` is a MongoDB modifier that performs the incrementing action inside the MongoDB server, essentially allowing us to outsource the calculation. To use it, we pass a document (object) alongside it containing the key to increment and the amount to increase it by. So we pass `votes` and 1.

`$inc` will create the `votes` field if it doesn't exist, and increment it by one (we can also decrement using minus figures). Next are the options to be passed to MongoDB. We've set `safe` to `true`, which tells MongoDB to check that the command was successful and send any errors if it wasn't. For the callback to work correctly, `safe: true` must be passed, otherwise errors are not caught, and the callback occurs immediately.



Upserting

Another useful option we can set is `upsert: true`. This is a really convenient MongoDB feature that either updates a record or inserts it if it doesn't exist.

Inside the `update` callback we run a chain of `find`, `sort`, `limit`, `each`, `find`, without any parameters, which returns all our records. `sort` requires keys and a positive or negative 1, indicating ascending or descending. `limit` takes an integer of maximum records, and `each` loops through our records. Inside the `each` callback, we output every `author`, `quote`, and `votes` of doc, closing the connection when no docs are left.

See also

- ▶ *Connecting and sending SQL to a MySQL server* discussed in this chapter
- ▶ *Storing and retrieving data with Mongoskin* discussed in this chapter
- ▶ *Storing data to CouchDB with Cradle* discussed in this chapter

Storing and retrieving data with Mongoskin

Mongoskin is a convenience library providing a high-level interface for `mongodb` without blocking access to the existing `mongodb` methods.

For this recipe, we'll re-implement the `quotes` database in MongoDB using Mongoskin.

Getting ready

We'll need the `mongoskin` module.

```
npm install mongoskin
```

We can also create a new folder, with a new `quotes.js` file.

How to do it...

We'll require `mongoskin` and use it to make a `client` and a `collection` instance. We don't need to create a `server` instance, nor manually open the client as in the former recipe, `mongoskin` takes care of that.

```
var mongo = require('mongoskin');
var client = mongo.db('localhost:27017/quotes');
var collection = client.collection('quotes');
var params = {author: process.argv[2], quote: process.argv[3]};
```

As in the previous recipe, we've defined our `params` object for user input.

Instead of requiring us to use JavaScript's potentially human error prone `new` keyword, `mongoskin` provides a builder method (`mongo.db`), allowing us to define our host, port, and database name with the familiar URI schema.



See <http://www.yuiblog.com/blog/2006/11/13/javascript-we-hardly-new-ya/> for why the new prefix could be considered error prone.

Since we don't need to open our client (`mongoskin` opens it for us), we can just go ahead and implement our `insert` and `find` operations:

```
if (params.author && params.quote) {
  collection.insert({author: params.author, quote: params.quote});
}
if (params.author) {
  collection.findEach({}, function (err, doc) {
    if (err) { throw err; }
    if (doc) { console.log('%s\n', doc.author, doc.quote);
      return; }
    client.close();
  });
  return;
}
client.close();
```

And we're done.

How it works...

We transparently connect to our database using `Mongoskin's db` method, and we're immediately able to grab our collection.

As in our previous recipe, we check for the `author` and `quote` command-line arguments, and then call the `mongodb insert` method which is inherently available through the `mongoskin` module.

Next, after checking for `author`, we use the `findEach` method of `mongoskin`. The `findEach` method wraps `collection.find.each` as used in the previous recipe.

Within `findEach` we output the `author` and `quote` properties of each `doc` to the console. When there's no documents left, we explicitly `close` the `client` connection.

There's more...

`Mongoskin` does an excellent job of making our lives easier. Let's look at another `Mongoskin` feature that simplifies our interactions with `MongoDB`.

Collection binding

Mongoskin provides a `bind` method that makes a collection available as a property of our `client` object. So if we did:

```
client.bind('quotes');
```

We could access the `quotes` collection via `client.quotes`.

This means we can discard the `collection` variable and use binding instead. The `bind` method also accepts an object of methods, which then applies to the bound collection. For instance, if we defined a method called `store`, we would access it as follows:

```
client.quotes.store(params.author, params.quote);
```

So let's create a new file called `quotes_bind.js` to re-implement `quotes.js` in Mongoskin, this time using the collection binding approach.

We'll begin with our top-level variables:

```
var mongo = require('mongoskin');
var client = mongo.db('localhost:27017/quotes');
var params = {author: process.argv[2], quote: process.argv[3]};
```

We have no need for a `collection` variable since we'll be accessing our collection via `bind`.

Now let's define a `store` method for inserts and a `show` method for displaying quotes:

```
client.bind('quotes', {
  store: function (author, quote) {
    if (quote) { this.insert({author: author, quote: quote}); }
  },
  show: function (author, cb) {
    this.findEach({author: author}, function (err, doc) {
      if (err) { throw err; }
      if (doc) { console.log('%s: %s \n', doc.author, doc.quote);
      return; }
      cb();
    });
  }
});
```

Then our logic to interact with our new bound methods:

```
client.quotes.store(params.author, params.quote);

if (params.author) {
  client.quotes.show(params.author, function () {
    client.close();
  });
  return;
}

client.close();
```

The `bind` method of Mongoskin painlessly abstracts involved database operations into an easy dot-notation format.

 We baked some of our `params` checking functionality into our `store` method, only calling `insert` if a quote exists. In all our recipes, we only needed to check for the second argument (`params.quote`), we couldn't have `params.quote` without `params.author`. Both arguments were checked in the prior recipes to demonstrate how it might work in other scenarios (for instance, if we received our parameters via a POST request).

See also

- ▶ *Storing and retrieving data with MongoDB* discussed in this chapter
- ▶ *Storing data to CouchDB with Cradle* discussed in this chapter
- ▶ *Retrieving data from CouchDB with Cradle* discussed in this chapter

Storing data to CouchDB with Cradle

In order to achieve stellar performance speeds, MongoDB has a relaxed view towards ACID (Atomicity Consistency Isolation Durability) compliance. However, this means there is an (unlikely) chance that data can become corrupted (especially if there was a power cut in the middle of an operation). CouchDB, on the other hand, is ACID compliant to the extent that, when replicated and synchronized, data eventually becomes consistent. Therefore, while slower than MongoDB, it has the added reliability advantage.

CouchDB is entirely administrated via HTTP REST calls, so we could do all of our work with CouchDB using `http.request`. Nevertheless, we can use Cradle to interact with CouchDB in an easy, high-level way, along with the added speed enhancement of automated caching.

In this recipe, we'll use Cradle to store famous quotes to CouchDB.

Getting ready

We'll need to install and run CouchDB, we can head on to <http://wiki.apache.org/couchdb/Installation> for instructions on how to install for your particular operating system.

After installation, we can check if CouchDB is running and access the Futon administration panel by pointing our browser to `http://localhost:5984/_utils`.

We'll also need the `cradle` module.

```
npm install cradle@0.6.3
```

And we can create a new folder with a new `quotes.js` file in it.

How to do it...

First, we require `cradle` and load our quotes database, creating it if necessary. We'll also define an error-handling function and our `params` object for easy command-line interaction:

```
var cradle = require('cradle');
var db = new(cradle.Connection)().database('quotes');
var params = {author: process.argv[2], quote: process.argv[3]};
function errorHandler(err) {
  if (err) { console.log(err); process.exit(); }
  // checkAndSave function here
```

Before we can write to our database, we need to know it exists:

```
db.exists(function (err, exists) {
  errorHandler(err);
  if (!exists) { db.create(checkAndSave); return; }
  checkAndSave();
});
```

Notice that we pass in `checkAndSave` as the callback of `db.create`, the following function goes above the `db.exists` invocation:

```
function checkAndSave(err) {
  errorHandler(err);

  if (params.author && params.quote) {
    db.save({author: params.author, quote: params.quote},
    errorHandler);

  }
}
```

The `err` parameter we are handling in `checkAndSave` would have been passed in from `db.create`.

How it works...

CouchDB is administrated via HTTP requests, but Cradle provides an interface for making these requests. When we invoke `db.exists`, Cradle sends a HEAD request to `http://localhost:5984/quotes` and checks whether the reply status is 404 Not Found or 200 OK. We could perform the same check with command-line program's `curl` and `grep` as follows:

```
curl -I http://localhost:5984/quotes | grep -c "200 OK"
```

This would output 1 if the database exists, and 0 if it does not. If our database doesn't exist, we call the `db.create` method of `cradle` which sends an HTTP PUT request to the CouchDB server. In `curl` this would be:

```
curl -X PUT http://localhost:5984/quote
```

We pass in our `checkAndSave` function as the callback of `db.create`, or we call it from the callback of `db.exists` if the database does exist. This is essential. We cannot save data to a database that doesn't exist and we have to wait for the HTTP response before we know whether it exists (or whether it has been created).

`checkAndSave` looks for command-line arguments, and then saves the data accordingly. For instance, if we ran the following from the command line:

```
node quotes.js "Albert Einstein" "Never lose a holy curiosity."
```

`checkAndSave` would realize there are two parameters passing these as `author` and `quote` to `db.save`. Cradle would then POST the following, with `Content-Type` set to `application/json`:

```
{"author": "Albert Einstein", "quote": "Never lose a holy curiosity"}
```

On top of this, Cradle adds a caching layer, which in our example is of little use since caching data is lost whenever our application exits. However, in server implementation, the caching would become very useful in answering similar requests quickly and efficiently.

There's more...

Couch stands for **Cluster Of Unreliable Commodity Hardware**, let's take a brief look at the clustering side of CouchDB.

Scaling CouchDB with BigCouch

Scaling is about making your application responsive to an anticipated demand, but different projects have different characteristics. Therefore, each scaling venture requires an individualized approach.

If a web service was heavily built around database interaction, scaling the database layer would be a priority when responding to changes in service demand. Scaling CouchDB (or anything else) can be a very in-depth procedure, necessarily so for specialized projects.

However, the Open Source BigCouch project has an ability to scale CouchDB in a transparent and generic fashion. With BigCouch, we can scale CouchDB across servers, but interact with it as if it was on one server. BigCouch can be found at <https://www.github.com/cloudant/bigcouch>.

See also

- ▶ *Retrieving data from CouchDB with Cradle* discussed in this chapter
- ▶ *Storing and retrieving data with MongoDB* discussed in this chapter
- ▶ *Storing and retrieving data with Redis* discussed in this chapter

Retrieving data from CouchDB with Cradle

CouchDB doesn't use the same query paradigm that MySQL and MongoDB subscribe to. Instead, it uses a pre-created view to retrieve the desired data.

In this example, we'll use Cradle to obtain an array of quotes according to the specified author, outputting our quotes to the console.

Getting ready

As in the previous recipe, *Storing data to CouchDB with Cradle*, we'll need CouchDB installed on our system, along with `cradle`. We can also take the `quotes.js` file from that recipe, placing it in a new directory.

How to do it...

We're working on the `quotes.js` file from the prior task where we called `checkAndSave` if our database existed, or we called it from the callback of `db.create` if it didn't exist. Let's modify `checkAndSave` slightly as shown in the following code:

```
function checkAndSave(err) {
  errorHandler(err);
  if (params.author && params.quote) {
    db.save({author: params.author, quote: params.quote},
    outputQuotes();
    return;
  }
  outputQuotes();
}
```

We've added a new function invocation, `out put Quotes`, to the end of `checkAndSave` and also as the callback of `db.save`. `out put Quotes` is going to access a special `CouchDB_design` document called a view.

Before we look at `out put Quotes`, let's look into another new function we'll be creating called `createQuotesView`. It should be placed just under `errorHandler` but above the rest of the code as follows:

```
function createQuotesView(err) {
  errorHandler(err);
  db.save('_design/quotes', {
    views: { byAuthor: { map: 'function (doc) { emit(doc.author, doc)' } },
    }, out put Quotes);
}
```

`createQuotesView` also calls the `out put Quotes` function from the `db.save` callback parameter. `out put Quotes` is now called from three places: the `db.save` callback of `checkAndSave`, the end of `checkAndSave`, and in the `db.save` callback of `createQuotesView`.

Let's take a look at `out put Quotes`:

```
function out put Quotes(err) {
  errorHandler(err);

  if (params.author) {
    db.view('quotes/byAuthor', {key: params.author},
      function (err, rowsArray) {
        if (err && err.error === "not_found") {
          createQuotesView();
          return;
        }
        errorHandler(err);

        rowsArray.forEach(function (doc) {
          console.log('%s %s \n', doc.author, doc.quote); return;
        });
      });
  }
}
```

`out put Quotes` goes before `checkAndSave`, but after `createQuotesView`.

How it works...

The key to querying a CouchDB database is views. There are two types of views: permanent and temporary. In `createQuot esVi ew` we define a permanent view using `db.save`, setting the document ID to `_des i gn/quot es`. We then define a `views` field containing an object called `byAut hor` which holds a key called `map` whose value is a string formatted function.

Temporary views would be stored with an ID of `quot es/_temp_v i ew`. However, these should only be used for testing,. They're very expensive computationally and shouldn't be used for production.

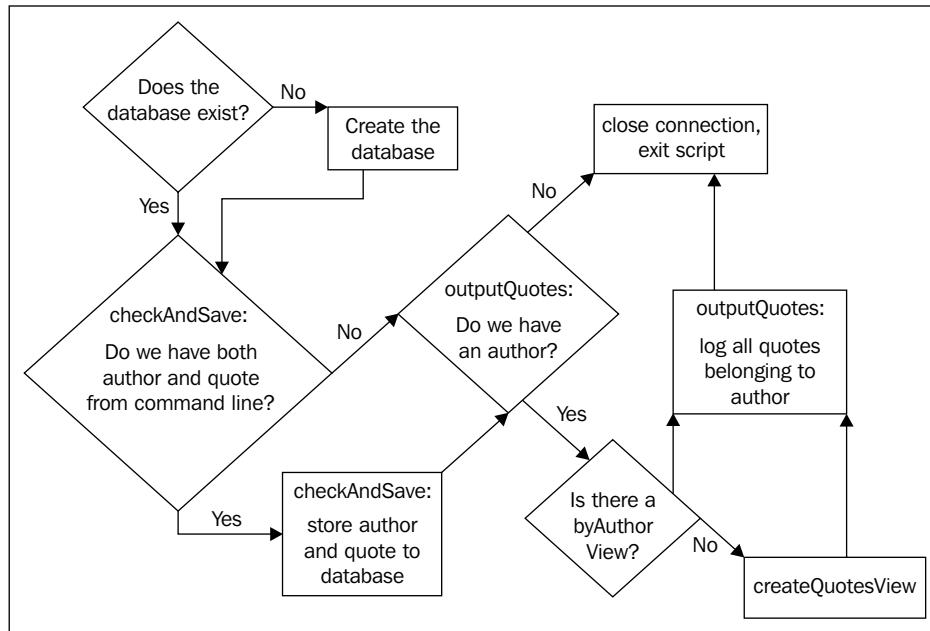
The mapping function is string formatted because it's passed to CouchDB via an HTTP request. CouchDB `map` functions are not executed with Node, they run within the CouchDB server. A `map` function defines the query we wish to run on the database through the CouchDB servers `emit` function. The first argument of `emit` specifies which field to query, (in our case `doc.aut hor`), and the second specifies what to output as a result of the query (we want the whole doc). If we wanted to search for Albert Einstein, we would make a GET request to: `http://localhost:5984/quot es/_des i gn/quot es/_vi ew/byAut hor?key="Albert Einstein"`.

Cradle provides a shorthand method for this request, `db.v i ew`, which appears in our `out put Quot es` function. `db.v i ew` allows us to simply pass in `quot es/byAut hor` with a second object containing the `key` parameter (that is, our query), essentially filling in the special underscore routes for us.

`db.v i ew` parses the incoming JSON and provides it via the second parameter of its callback, which we named `rows Array`. We loop through the array using `for Each` and finish off by outputting `aut hor` and `quot e` to the console, as in previous recipes.

However, before we loop through the array we need to check that our view actually exists. Views only need to be generated once. After this, they are stored in the CouchDB database. Therefore, we don't want to create a view every time we run our application. So when we call `db.v i ew` we look to see if a `not_f ound` error occurs in the `db.v i ew` callback. If our view isn't found, we call `creat eQuot esVi ew`.

In broad terms, the process goes something like this:



There's more...

CouchDB is great for getting to grips right out of the box. However, there are certain security considerations we must be attuned to before deploying a CouchDB-backed app to the web.

Creating an admin user

CouchDB requires no initial authorization settings, which is fine for development. However, as soon as we expose CouchDB to the outside world, anyone on the Internet has permission to edit our entire database: data designs, configuration, users, and so on.

So before deploying, we want to set a username and password. We can achieve this with the `_config` API:

```
curl -X PUT http://localhost:5984/_config/admins/dave -d '{"cookit": "password"}'
```

We have created the admin user `dave` and set the password to `cookit`. Now the right to certain calls will be denied without authentication, including the creation or deletion of databases, modifying design documents (for example, for views), or access to the `_config` API.

For instance, say we wanted to view all admin users, we could say:

```
curl http://localhost:5984/_config/admins
```

CouchDB will reply:

```
{"error": "unauthorized", "reason": "You are not a server admin."}
```

However, if we include authentication information:

```
curl http://dave:cookit@localhost:5984/_config/admins
```

We get our only admin user along with a hash of his password:

```
{"dave": "-hashed-42e68653895a4c0a5c67baa3cfb9035d01057b0d,44c62ca1bfd4872b773543872d78e950"}
```

Using this method to remotely administer a CouchDB database is not without its security flaws. It forces us to send passwords as plain text over non-secure HTTP. Ideally, we need to host CouchDB behind an HTTPS proxy so the password becomes encrypted as it's sent. See the recipe *Setting up an HTTPS server* discussed in *Chapter 7, Implementing Security, Encryption, and Authentication*.

If CouchDB is behind HTTPS, cradle can connect to it as follows:

```
var db = new (cradle.Connection)({secure: true,
                                 auth: {username: 'dave',
                                         password: 'cookit'}})
      .database('quotes');
```

We pass an options object when we create our connection. The secure property tells cradle we are using SSL, and auth contains a sub-object with login details.

Alternatively, we create a Node app to authenticate with a local CouchDB instance (so that no password is sent to an external HTTP address) and act as a layer between external requests and CouchDB.

Locking all modifying operations to an admin user

Even if an admin user is set, unauthenticated users still have permission to modify existing databases. If we were only writing to CouchDB server side (but reading from either server or clients), we could lock all write operations for non-admin users with a validation function.

A validation function is written in JavaScript and runs on the CouchDB server (like a map function). Once a validation function is defined, it's executed against all user input for the database it is applied to. Three objects appear as parameters in the function: the new document (newDoc), the previous currently stored document (savedDoc), and the user context (userCtx), which holds authenticated user information.

Within a validation function we can examine and qualify these objects, calling CouchDB's t hr ow function to reject operation requests that fail to meet our requirements.

Let's make a new file called `database_lockdown.js`, and begin by connecting to our database:

```
var cradle = require('cradle');
var db = new (cradle.Connection)({auth:
  {username: 'dave',
  password: 'cookit'}})
  .database('quotes');
```

We pass in an `options` object to the new cradle connection. It contains authentication information which will now be necessary to create a validation function if we have set a new admin user according to the previous subsection *Creating an admin user*.

Let's create our validation function, and save it as a `_design` document:

```
var admin_lock = function (newDoc, savedDoc, userCtx) {
  if (userCtx.roles.indexOf('_admin') === -1) {
    throw({unauthorized: 'Only for admin users'});
  }
}
db.save('_design/_auth', {
  views: {},
  validate_doc_update: admin_lock.toString()
});
```

As soon as we execute:

```
node database_lockdown.js
```

All write-related operations will now require authorization.

Like views, we store validation functions within a document that has a `_design/` prefixed ID. The other part of the ID can be anything, but we named it `_auth` which reflects conventional practice when a validation function serves this type of purpose. The field name, though, must be called `validate_doc_update`.

By default Cradle assumes any `_design` documents passed to `db.save` are a view. In order to prevent Cradle from wrapping our `validate_update_doc` field into a view, we specify an empty object to the `views` property.

`validate_update_doc` must be passed a string-formatted function, so we define our function under the `admin_lock` variable and call `toString` on it as it's passed into `db.save`.

`admin_lock` is never intended for execution by Node. It's an aesthetic approach to constructing our function before passing it to CouchDB.

When an operation occurs on the database, our `admin_lock` function (which becomes CouchDB's `validate_update_doc` function) asks CouchDB to check if the user requesting the operation has the `_admin` user role. If not, it tells CouchDB to throw an unauthorized error thus denying access.

Exposing the CouchDB HTTP interface to remote connections

By default, CouchDB binds to `127.0.0.1`. This ensures that only local connections can be made to the database ensuring safety prior to security enforcements. Once we have CouchDB set up behind HTTPS with at least one admin user set, we can bind CouchDB to `0.0.0.0` which makes the REST interface accessible via any IP address. This means that remote users can access our CouchDB HTTP interface via our server's public IP address, or more likely via our server's domain name. We can set the bind address with `_config` as follows:

```
curl -X PUT https://u:p@localhost:5984/_config/httpd/bind_address -d '"0.0.0.0"'
```

Where `u` and `p` are the admin username and password, respectively.

See also

- ▶ *Storing data to CouchDB with Cradle* discussed in this chapter
- ▶ *Storing and retrieving data with MongoDB* discussed in this chapter
- ▶ *Setting Up and HTTPS Web Server* discussed in Chapter 7, *Implementing Security, Encryption, and Authentication*

Accessing CouchDB changes stream with Cradle

One of CouchDB's most noteworthy features is the `_changes` API. With it we can view all alterations to a database via HTTP.

For instance, to see all changes made to our `quotes` database we can make a GET request to `http://localhost:5984/quotes/_changes`. Even better, if we wanted to hook up to a live stream, we would add the query parameter `?feed=continuous`.

Cradle provides an attractive interface to the `_changes` API, which we'll explore in this recipe.

Getting ready

We'll need a functioning CouchDB database and a way to write to it. We can use the `quotes.js` example used in *Storing data to CouchDB with Cradle*, so let's copy that into a new directory and then create a file alongside it called `quotes_stream.js`.

If we followed the *Creating an admin user* and *Locking all modifying operations to an admin user* sections of the previous recipe's *There's more...* section, we will need to modify the second line of `quotes.js` in order to continue to insert quotes in our database:

```
var db = new (cradle.Connection)({ auth: { username: 'dave',
                                             password: 'cookit' } })
    .database('quotes');
```

Where `dave` and `cookit` are the example username and password.

How to do it...

We require `cradle` and make a connection to our `quotes` database. Our stream is intended for use with a pre-existing database, so we won't be checking for database existence.

```
var cradle = require('cradle');
var db = new (cradle.Connection)().database('quotes');
```

Next, we call the `changes` method of `cradle` and listen to its `response` event, in turn listening to the passed in `response` emitter's `data` event:

```
db.changes().on('response', function (response) {
    response.on('data', function (change) {
        var changeIsObj = {}.toString.call(change) === '[object Object]';
        if (change.deleted || changeIsObj) { return; }
        db.get(change.id, function (err, doc) {
            if (!doc) { return; }
            if (doc.author && doc.quote) {
                console.log('%s %s\n', doc.author, doc.quote);
            }
        });
    });
});
```

To test our `changes` stream implementation we'll open two terminals. In one we'll run the following command:

```
node quotes_stream.js
```

Interfacing with Databases

In the other terminal window we can add some quotes using `quotes.js`:

```
node quotes.js "Yogi Berra" "I never said most of the things I said"
```

```
node quotes.js "Woody Allen" "I'd call him a sadistic hippophilic  
necrophile, but that would be beating a dead horse"
```

```
node quotes.js "Oliver Wendell Holmes" "Man's mind, once stretched by a  
new idea, never regains its original dimensions"
```

The screenshot shows two terminal windows side-by-side. The left terminal window has a title bar 'Node Cookbook' and contains the command 'node@cookbook~\$ node quotes_stream.js'. It displays four quotes added sequentially: 'Albert Einstein: Never lose a holy curiosuty', 'Yogi Berra: I never said most of the things I said', 'Woody Allen: I'd call him a sadistic hippophilic necrophile, but that would be beating a dead horse', and 'Oliver Wendell Holmes: Man's mind, once stretched by a new idea, never regains its original dimensions'. The right terminal window also has a title bar 'Node Cookbook' and contains the command 'node@cookbook~\$ node quotes.js'. It lists the same four quotes, indicating they were added to a stream. Both terminals have a standard OS X interface with a dark background.

As each new quote is added in the left-hand terminal, it appears in the right.

`quotes_stream.js` was opened up before any new quotes were added, and immediately displayed the `Albert Einstein` quote which was added in the *Storing data to CouchDB with Cradle* recipe. After this, new quotes appeared in the stream as they were added.

How it works...

The `changes` method can be passed a callback, which simply returns all changes up to the present and then exits. If we do not pass a callback to `changes` it adds the `?feed=continuous` parameter to the HTTP CouchDB REST call and returns `Event Emitter`. CouchDB then returns a streamed HTTP response to Cradle which is sent through as the `response` parameter of the `response` event. The `response` parameter is also `Event Emitter`, and we listen for changes via the `data` event.

On each `data` event, the callback handles the `change` parameter. Two data events are fired for each change, one is a JSON string, the other is a JavaScript object containing the equivalent JSON data. We check that the `change` parameter's type is an object (`changeIsObj`) before proceeding. The `change` object holds metadata for our database entries. It has a sequence number (`change.seq`), a revision number (`change.changes[0].rev`), it sometimes contains a deleted property (`changes.deleted`), and always has an `id` property.

If the `deleted` property is found, we need to `return` early as `db.get` can't fetch a deleted record. Otherwise, we pass `change.id` into `db.get`, which provides access to a document ID. The `doc` is passed into the callback of `db.get`. We only want to output changes regarding our quotes, so we check for `author` and `quote` fields and log them to the console.

See also

- ▶ *Storing data to CouchDB with Cradle* discussed in this chapter
- ▶ *Retrieving data from CouchDB with Cradle* discussed in this chapter
- ▶ *Implementing PubSub with Redis* discussed in this chapter

Storing and retrieving data with Redis

Redis is a non-traditional database, dubbed a data structures server, which functions in operational memory with blazingly fast performance.

Redis is excellent for certain tasks, as long as the data model is fairly simple and isn't so large that it swamps your server RAM. Good examples of where Redis shines are in site analytics, server-side session cookies, and providing a list of logged-in users in real time.

In the spirit of our theme, we will re-implement our quotes database with Redis.

Getting ready

We'll be using the `node_redis` client.

```
npm install redis
```

We also need to install the Redis server, which can be downloaded from <http://www.redis.io/download> along with installation instructions.

Let's also create a new directory with a new `quotes.js` file.

How to do it...

Let's create the `redis` module, create a connection, and listen for the `ready` event emitted by the `redis client`, not forgetting to load the command-line arguments into the `params` object.

```
var redis = require('redis');
var client = redis.createClient();
var params = {author: process.argv[2], quote: process.argv[3]};

client.on('ready', function () {
    // quotes insertion and retrieval code to go here...
});
```

Next, we'll check for `author` and `quote` via the command line. If they're defined, we'll insert these as hash (an object structure) into Redis, inside our `ready` event callback:

```
if (params.author && params.quote) {
    var randKey = "Quotes:" + (Math.random() * Math.random())
        .toString(16).replace('.', '');

    client.hset(randKey, {"author": params.author,
                          "quote": params.quote});

    client.sadd('Author:' + params.author, randKey);
}
```

Not only did we add our data to Redis, we also constructed a basic index on the fly enabling us to search for quotes by author in our next piece of code.

We check for the existence of the first command-line argument, the `author`, and then output quotes by that author.

```
if (params.author) {
    client.smembers('Author:' + params.author, function (err, keys) {
        keys.forEach(function (key) {
```

```

        client.hgetAll(key, function (err, hash) {
            console.log(`%s: %s \n`, hash.author, hash.quote);
        });
    client.quit();
});
return;
}
client.quit();

}); // closing brackets of the ready event callback function

```

How it works...

If both `author` and `quote` are specified via the command line, we go ahead and generate a random key prefixed with `Quote:`. So each key will look something like `Quote:08d780a57b035f`. This helps us to identify keys in debugging and it's also common convention to prefix the Redis keys with names delimited by a colon.

We pass this key into `client.hmset`, a wrapper for the Redis `HSET` command, which allows us to create multiple hashes. Unlike the raw `HSET`, `client.hmset` also accepts a JavaScript object (instead of an array) to create multiple key assignments. With the standard Redis command-line client, `redis-cli`, we would have to say:

```
HSET author "Steve Jobs" quote "Stay hungry, stay foolish."
```

We could hold to this format by using an array containing keys next to values, but an object seems kinder and more familiar to the eyes of a JavaScript folk.

Every time we store a new quote with `client.hmset`, we add the `randKey` for that quote to the relevant author set via the second parameter of `client.sadd`. `client.sadd` allows us to add a member to a Redis set (a set is like an array of strings). The key for our `SADD` command is based on the intended author. So in the Steve Jobs quote used above, the key to pass into `client.sadd` would be `Author:Steve Jobs`.

Next, if an author is specified we perform an `SMEMBERS` using `client.smembers`. This returns all the values we stored to a specific authors' set, being the keys for all quotes relating to that author.

We loop through these keys using `forEach`, passing every key into `client.hgetall`. Redis `HGETALL` returns hash (object) that we passed into `client.hmset` earlier. Each author and quote is then logged to the console, and `client.quit` gracefully exits our script once all Redis commands have been executed.

`client.quit` is also included at the end of the `ready` event on? occasions where no command-line arguments have been specified.

There's more...

Redis is a speed freak's dream, but we can still make optimizations.

Speeding up node Redis module

By default, the `redis` module uses a pure JavaScript parser. However, the Redis project provides a Node `hiredis` module: a C bindings module which binds to the official Redis client, Hiredis. Hiredis is faster (being written in C) than the JavaScript parser.

The `redis` module will interface with the `hiredis` module if it is installed. Therefore, we can achieve performance benefits simply by installing `hiredis`:

```
npm install hiredis
```

Overcoming network latency by pipelining commands

Redis can receive multiple commands at once. The `redis` module has a `multi` method which sends collated commands en masse. If the latency (time taken for data to travel) was 20 ms per command, for 10 combined commands we save 180 ms ($10 \times 20 - 20 = 180$).

If we copy `quotes.js` to `quotes_multi.js` we can alter it accordingly:

```
// top variables, client.ready...

if (params.author && params.quote) {
    var randKey = "Quote:" + (Math.random() * Math.random())
        .toString(16).replace('.', '');

    client.multi()
        .hmset(randKey, {"author": params.author,
                         "quote": params.quote})
        .sadd('Author:' + params.author, randKey)
        .exec(function (err, replies) {
            if (err) { throw err; }
            if (replies[0] == "OK") { console.log('Added...\n'); }
        });
}

//if params.author, client.smembers, client.quit
```

We can see our original Redis commands highlighted, only they have been chained with `client.multi`. Once all commands have been added to `client.multi`, we invoke its `exec` method. Finally, we use the callback of `exec` to verify that our data was successfully added.

We didn't provision SMEMBERS for pipelining. SMEMBERS must be called after the quote has been added, or else the new quote won't be displayed. If SMEMBERS was combined with HMSET and SADD it would be executed asynchronously alongside them. There's no guarantee that the new quote would be available to SMEMBERS. In fact, it's unlikely since SMEMBERS is more complex than SADD, so it takes longer to process.

See also

- ▶ *Storing and retrieving data with Mongoskin* discussed in this chapter
- ▶ *Connecting and sending SQL to a MySQL server* discussed in this chapter
- ▶ *Implementing PubSub with Redis* discussed in this chapter

Implementing PubSub with Redis

Redis exposes a Publish-Subscribe messaging pattern (not so dissimilar to the CouchDB changes stream), which can be used to listen to specific data change events. Data from these events could be passed between processes to, for instance, instantly update a web app with fresh new data.

With PubSub, we publish a message to a specific channel, this channel can then be picked up by any amount of subscribers. The publishing mechanism doesn't care who's listening or how many, it chats away regardless.

In this recipe, we will create a publishing process and a subscribing process. For publishing, we'll extend our quotes.js file from the previous recipe *Storing and retrieving data with Redis*, and we'll write the code to a new file for the subscription mechanism.

Getting ready

Let's create a new directory, copy quotes.js from the previous recipe, and rename it to quotes_publish.js. We will also create a file called quotes_subscribe.js. We'll need to ensure that Redis is running. If it isn't installed and running globally, we can navigate to the directory Redis was unpacked to and run ./redis-server from the src folder.

How to do it...

In quotes_publish.js we add one extra line of code inside our first conditional statement, just after our client.sadd call.

```
if (params.author && params.quote) {
    var randKey = "Quote:" + (Math.random() * Math.random())
        .toString(16).
    replace('.', '');
```

```
client.hmset(randKey, {"author": params.author,
                      "quote": params.quote});

client.sadd('Author:' + params.author, randKey);

client.publish(params.author, params.quote);

}
```

This means every time we add an author and quote, we publish the quote to a channel named after the author. We subscribe to channels using `quotes_subscribe.js`, so let's code it.

First, it must require the `redis` module and create a client:

```
var redis = require('redis');
var client = redis.createClient();
```

We're going to provide the option to subscribe to multiple channels, again using the command line as our elementary input method. To achieve this, we'll loop through `process.argv`:

```
process.argv.slice(2).forEach(function (authorChannel, i) {

  client.subscribe(authorChannel, function () {
    console.log('Subscribing to ' + authorChannel + ' channel');
  });

});
```

Now that we are subscribing to channels, we need to listen to messages:

```
client.on('message', function (channel, msg) {
  console.log("\n%: %", channel, msg);
});
```

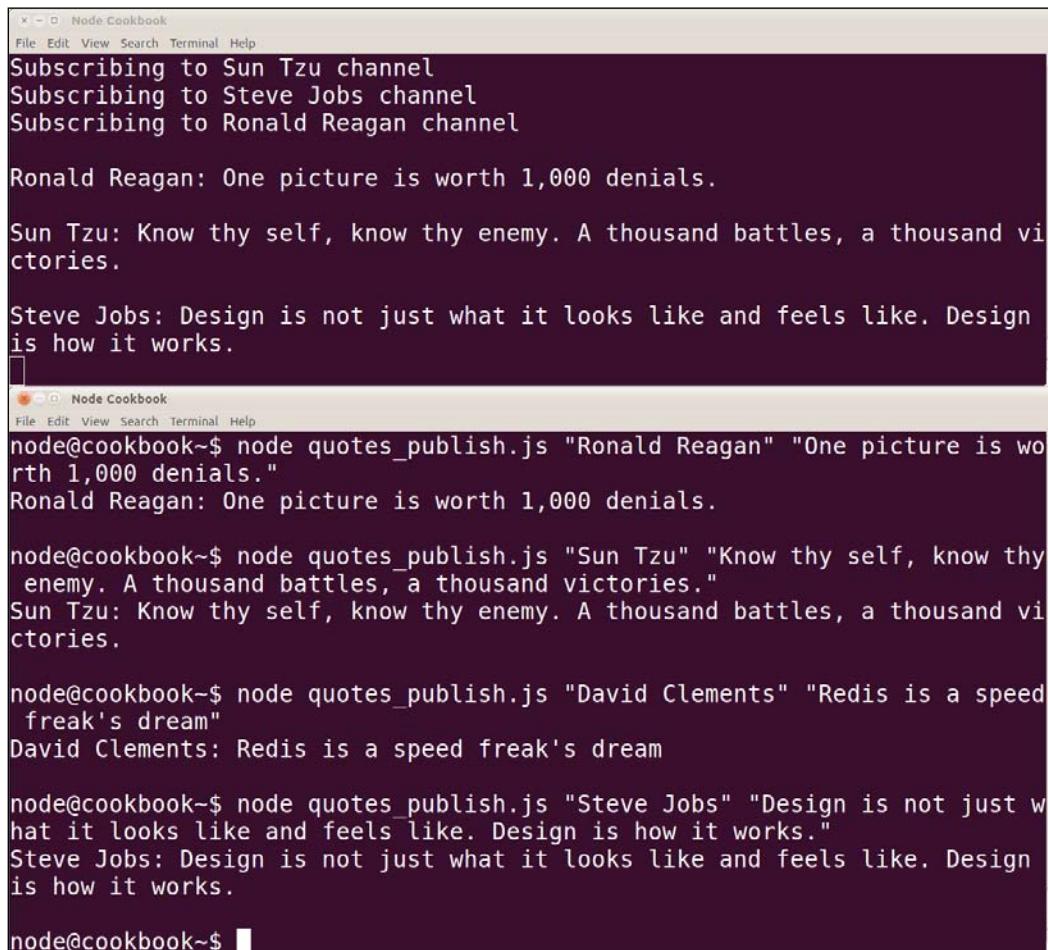
We can test our PubSub functionality by first running `quotes_subscribe.js` along with some specified authors:

```
node quotes_subscribe.js "Sun Tzu" "Steve Jobs" "Ronald Reagan"
```

Then we open a new terminal and run several authors and quotes through `quotes_publish.js`.

```
node quotes_publish.js "Ronald Reagan" "One picture is worth 1,000
denials."
node quotes_publish.js "Sun Tzu" "Know thy self, know thy enemy. A
thousand battles, a thousand victories."
node quotes_publish.js "David Clements" "Redis is a speed freak's dream"
node quotes_publish.js "Steve Jobs" "Design is not just what it looks
like and feels like. Design is how it works."
```

Let's see it in action:



The image shows two terminal windows side-by-side. Both windows have a title bar 'Node Cookbook' and a menu bar 'File Edit View Search Terminal Help'. The left terminal window displays the following output:

```
Subscribing to Sun Tzu channel
Subscribing to Steve Jobs channel
Subscribing to Ronald Reagan channel

Ronald Reagan: One picture is worth 1,000 denials.

Sun Tzu: Know thy self, know thy enemy. A thousand battles, a thousand vi
ctories.

Steve Jobs: Design is not just what it looks like and feels like. Design
is how it works.
```

The right terminal window displays the following output:

```
node@cookbook~$ node quotes_publish.js "Ronald Reagan" "One picture is wo
rth 1,000 denials."
Ronald Reagan: One picture is worth 1,000 denials.

node@cookbook~$ node quotes_publish.js "Sun Tzu" "Know thy self, know thy
enemy. A thousand battles, a thousand victories."
Sun Tzu: Know thy self, know thy enemy. A thousand battles, a thousand vi
ctories.

node@cookbook~$ node quotes_publish.js "David Clements" "Redis is a speed
freak's dream"
David Clements: Redis is a speed freak's dream

node@cookbook~$ node quotes_publish.js "Steve Jobs" "Design is not just w
hat it looks like and feels like. Design is how it works."
Steve Jobs: Design is not just what it looks like and feels like. Design
is how it works.

node@cookbook~$
```

Only the channels we subscribed to appeared on the `quotes_subscribe.js` terminal.

How it works...

We access the Redis PUBLISH command in `quotes_publish.js` via `client.publish`, setting the channel name by author name.

In `quotes_subscribe.js` we loop through any arguments given via command line. (We apply `forEach` to `process.argv.slice(2)`. This removes the first two elements of the `process.argv` array, which would hold the command (`node`) and the path to our script. Each relevant argument is passed to `client.subscribe`, telling Redis we wish to SUBSCRIBE to that channel.

When a message arrives due to a subscription, the `redis` module `client` will emit a `message` event. We listen for this event and pass the incoming channel and msg (which will be author and quote accordingly) to `console.log`.

There's more...

Finally, we'll take a look at Redis security.

Redis authentication

We can set authentication for Redis with the `redis.conf` file, found in the directory we installed Redis to. To set a password in `redis.conf`, we simply add (or uncomment) `requirepass ourpassword`.

Then we make sure that our Redis server points to the configuration file. If we are running it from the `src` directory, we would initiate with the following command:

```
./redis-server ./redis.conf
```

If we wanted to set a password quickly we could say:

```
echo "requirepass ourpassword" | ./redis-server -
```

We could set a password from within Node with the `CONFIG SET` Redis command:

```
client.config('SET', 'requirepass', 'ourpassword');
```

To authenticate with a Redis server within Node, we can use the `redis` module's `auth` method, before any other calls (that is, prior to `client.ready`).

```
client.auth('ourpassword');
```

The password has to be sent before any other commands. The `redis` module's `auth` function takes care of things like reconnections by pushing the password into the `redis` module's internal operations. Essentially, we can call `auth` at the top of our code and never concern ourselves with authentication for that script again.

Securing Redis from external connections

If there was no need for external connections to Redis, we could bind it to 127.0.0.1, inhibiting all external traffic.

We can achieve this with a configuration file, for instance `redis.conf`, and add (or uncomment):

```
bind 127.0.0.1
```

Then, if running from the `src` folder, initialize our Redis server with:

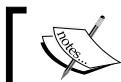
```
./redis-server ./redis.conf
```

Alternatively, we could do it as follows:

```
echo "bind 127.0.0.1" | ./redis-server -
```

Or in Node with the `redis` module's `config` method:

```
client.config('set', 'bind', '127.0.0.1');
```



If we installed Redis via a package manager it may already be configured to block external connections.



See also

- ▶ *Accessing CouchDB changes stream with Cradle* discussed in this chapter
- ▶ *Storing and retrieving data with Redis* discussed in this chapter

5

Transcending AJAX: Using WebSockets

In this chapter we will cover:

- ▶ Creating a WebSocket server
- ▶ Seamless fallbacking with `socket.io`
- ▶ Callbacks over `socket.io` transport
- ▶ Creating a real-time widget

Introduction

HTTP was not made for the kind of real-time web applications that many developers are creating today. As a result, all sorts of workarounds have been discovered to mimic the idea of bi-directional, uninterrupted communication between servers and clients.

WebSockets don't mimic this behavior, they provide it. A WebSocket works by stripping down an HTTP connection so it becomes a persistent TCP-like exchange, thus removing all the overhead and restrictions HTTP introduces.

The HTTP connection is stripped (or rather upgraded) when both the browser and server support WebSockets. The browser discovers this by communicating with the server via GET headers. Only newer browsers (IE10+, Google Chrome 14, Safari 5, Firefox 6) support WebSockets.

WebSockets is a new protocol. JavaScript combined with the Node framework is often versatile and low level enough to implement protocols from scratch, or failing that C/C++ modules can be written to handle more obscure or revolutionary logic. Thankfully, there's no need to write our own protocol implementation, the open source community has already provided.

In this chapter, we will be using some third-party modules to explore some of the potential of the powerful combination of Node and WebSockets.

Creating a WebSocket server

For this task, we will use the non-core `websocket` module to create a pure WebSocket server that will receive and respond to WebSocket requests from the browser.

Getting ready

We'll create a new folder for our project which will hold two files: `server.js` and `client.html`. `server.js`. They provide the server-side websocket functionality and serve up the `client.html` file. For the server-side WebSocket functionality, we also need to install the `websocket` module:

```
npm install websocket
```



For more information on the `websocket` module, see
<https://www.github.com/Worlize/WebSocket-Node>.

How to do it...

A WebSocket is an HTTP upgrade. As such, WebSocket servers run on top of HTTP servers. So we'll require the `http` and `websocket` servers, plus we'll also load our `client.html` file (which we'll be creating soon) and the `url` module:

```
var http = require('http');
var WSServer = require('websocket').server;
var url = require('url');
var clientHTML = require('fs').readFileSync('client.html');
```

Now let's create the HTTP server, and supply it to a new WebSocket server:

```
var plainHttpServer = http.createServer(function (request, response) {
  response.writeHead(200, {'Content-type': 'text/html'});
  response.end(clientHTML);
}).listen(8080);

var webSocketServer = new WSServer({httpServer: plainHttpServer});

var accept = [ 'localhost', '127.0.0.1' ];
```



We bind our HTTP server to port 8080 because binding to ports lower than 1000 takes root access. This means our script would have to be executed with root privileges which is a bad idea. See *Chapter 10, Taking It Live*, for more information and how to bind to the HTTP port (80) safely.



We've also created a new array, called `accept`. We use this inside the WebSocket server to restrict which originating sites can connect. For our example, we only allow connections from localhost or 127.0.0.1. If we were hosting live we would include any domains pointing to our server in the `accept` array.

Now that we have our `WebSocket Server` instance, we can listen to its `request` event and respond accordingly:

```
webSocketServer.on('request', function (request) {
  request.origin = request.origin || '*'; // no origin? Then use * as
  // wildcard.
  if (accept.indexOf(url.parse(request.origin).hostname) === -1) {
    request.reject();
    console.log('disallowed ' + request.origin);
    return;
  }

  var websocket = request.accept(null, request.origin);

  websocket.on('message', function (msg) {
    console.log('Received "' + msg.utf8Data + '" from ' + request.
    origin);
    if (msg.utf8Data === 'Hello') {
      websocket.send('WebSockets!');
    }
  });

  websocket.on('close', function (code, desc) {
    console.log('Disconnect: ' + code + ' - ' + desc);
  });
});
```

In our `request` event callback, we conditionally accept the request, then listen for the `message` and `close` events, responding with **WebSockets!** if the message from the client is `Hello`.

Now for the client, we'll place the following HTML structure:

```
<html>
<head>
</head>
<body>
<input id=msg><button id=send>Send</button>
<div id=output></div>

<script>
//client side JavaScript will go here
</script>

</body>
</html>
```

And the content of our `script` tags should look as follows:

```
<script>
(function () {
    var ws = new WebSocket("ws://localhost:8080"),
        output = document.getElementById('output'),
        send = document.getElementById('send');

    function logStr(eventStr, msg) {
        return '<div>' + eventStr + ':' + msg + '</div>';
    }

    send.addEventListener('click', function () {
        var msg = document.getElementById('msg').value;
        ws.send(msg);
        output.innerHTML += logStr('Sent', msg);
    });

    ws.onmessage = function (e) {
        output.innerHTML += logStr('Received', e.data);
    };

    ws.onclose = function (e) {
        output.innerHTML += logStr('Disconnected', e.code + '-' + e.type);
    };

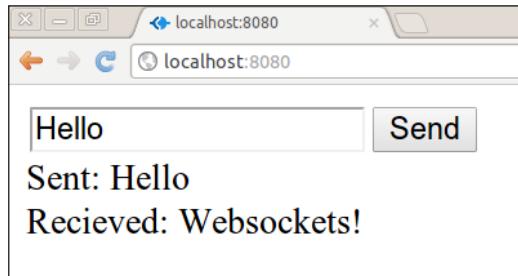
    ws.onerror = function (e) {
        output.innerHTML += logStr('Error', e.data);
    };
})();

</script>
```

If we initialize our server with `node server.js`, then direct our (WebSocket-compliant) browser to `http://localhost:8080`, type `Hello` in the textbox, and click on **Send**. The terminal console will output:

```
Received "Hello" from http://localhost:8080
```

And our browser will show that **Hello** was sent and **WebSockets!** was received, as shown in the following screenshot:



We could use our textbox to send any string we like to our server, but only **Hello** will gain a response.

How it works...

In `server.js`, when we require the `websocket` module's `server` method, we load a constructor function into `WSServer`, (which is why we capitalized the first letter). We initialize `WSServer` using `new` and pass in our `plainHttpServer` which transforms it into a WebSocket-enabled server.

The HTTP server will still serve normal HTTP requests, but when it receives a WebSocket connection handshake the `WebSocket Server` kicks into action to establish a persistent connection to the client.

As soon as the `client.html` file is loaded in the browser (served by the HTTP server in `server.js`) and the inline script is executed, the WebSocket upgrade request is made to the server.

When the server receives this WebSocket upgrade request, `WebSocket Server` emits a `request` event, which we scrutinize with our `accept` array before deciding whether we will respond.

Our `accept` array holds a whitelist of hosts we'll allow to interface with our WebSocket server. We gain some security by only allowing known sources to use our WebSocket server.

Inside the `WebSocket Server request` event, `request.origin` is parsed with `url.parse` to retrieve the host name section of the `origin` URL. If the host name isn't found in our `accept` whitelist we call `request.reject`.

If our originating host passes, we create a `websocket` variable from `request.accept`. The first parameter of `request.accept` allows us to define a custom sub-protocol. We could create an array of websockets using multiple `request.accepts` with different sub-protocols which represent different behaviors. When initiating our client, we would pass an additional argument containing that sub-protocol (for example, `new WebSocket("ws://localhost:8080", 'myCustomProtocol')`). However, we pass in `null`, such functionality isn't required for our purposes. The second parameter allows us to inform `request.accept` of the host we wish to allow (there is also a third that can be used for passing cookies).

For each message received from the client, `WebSocket` emits a `message` event. This is where we log the received data to the `console` and check whether the incoming message is `Hello`. If it is, we use the `WebSocket.send` method to respond to the client with **WebSockets!**.

Finally, we listen for the `close` event, to inform `console` that the connection has been terminated.

There's more...

WebSockets have so much potential for efficient, low latency real-time web apps, but compatibility can be a challenge. Let's look at some other uses for WebSockets, plus a tip on getting WebSockets to work in Firefox.

Supporting older Firefox browsers

Firefox versions 6 to 11 do support WebSockets. However, they use a vendor prefix so our `client.html` will not work on these Firefox versions.

To fix this, we simply prepend the following to the script in our `client.html` file:

```
window.WebSocket = window.WebSocket || window.MozWebSocket;
```

If the `WebSocket` API does not exist, we try `MozWebSocket`.

Creating a node-based WebSocket client

The `websocket` module also allows us to create a WebSocket client. We may wish to interface Node with a pre-existing WebSocket server, which is primarily for browser clients (if not, we are better off creating a simple TCP server. See *Chapter 8, Integrating Network Paradigms*).

So let's implement the same functionality in `client.html` using Node. We'll create a new file in the same directory, calling it `client.js`:

```
var WSClient = require('websocket').client;

new WSClient()

  .on('connect', function (connection) {
    var msg = 'Hello';

    connection.send(msg);
    console.log('Sent: ' + msg);

    connection.on('message', function (msg) {
      console.log("Received: " + msg.utf8Data);
    }).on('close', function (code, desc) {
      console.log('Disconnected: ' + code + ' - ' + desc);
    }).on('error', function (error) {
      console.log("Error: " + error.toString());
    });
  })

  .on('connectFailed', function (error) {
    console.log('Connect Error: ' + error.toString());
  })
  .connect('ws://localhost:8080/', null, 'http://localhost:8080');
```

For brevity, we've simply hardcoded our `msg` variable, though we could have used `process.stdin` or `process.argv` to input a custom message. We initialize a new client with the `websocket` module's `client` method. Then we immediately begin to listen for the `connect` and `connectFailed` events.

After the two `on` methods, we chain the `connect` method. The first parameter is our WebSocket server, the second is the protocol (remember, we have a `null` protocol for `request.accept` in our recipe), and the third defines the `request.origin` value.

Origin protection is designed to prevent an attack that only works from browsers. So although we can manufacture origins outside of the browser, it doesn't pose the same threat. The biggest threat is from a JavaScript injection attack into a high traffic site, that could cause a large amount of unauthorized connections from an unintended origin resulting in **Denial of Service**. See Chapter 7, *Implementing Security, Encryption, and Authentication*.

See also

- ▶ *Seamless fallbacking with socket.io* discussed in this chapter
- ▶ *Serving static files* Chapter 1, *Making a Web Server*

Seamless fallbacking with socket.io

Older browsers don't support WebSockets. So in order to provide a similar experience, we need to fall back to various browser/plugin-specific techniques to emulate WebSocket functionality to the best of the deprecated browser's ability.

Naturally, this is a mine field, requiring hours of browser testing and in some cases highly specific knowledge of proprietary protocols (for example, IE's Active X `htmlfile` object).

`socket.io` provides a WebSocket-like API to the server and client to create the best-case real-time experience across a wide variety of browsers, including old (IE 5.5+) and mobile (iOS Safari, Android) browsers.

On top of this, it also provides convenience features, such as disconnection discovery allowing for auto reconnects, custom events, namespacing, calling callbacks across the wire (see the next recipe *Callbacks over socket.io transport*), as well as others.

In this recipe, we will re-implement the previous task for a high compatibility WebSocket-type application.

Getting ready

We'll create a new folder with `newclient.html` and `server.js` files. We'll also install the `socket.io` module:

```
npm install socket.io
```

How to do it...

Like the `websocket` module, `socket.io` can attach to an HTTP server (though it isn't a necessity with `socket.io`). Let's create the `http` server and load `client.html`. In `server.js` we write:

```
var http = require('http');
var clientHtml = require('fs').readFileSync('client.html');

var plainHttpServer = http.createServer(function (request, response) {
    response.writeHead(200, {'Content-type' : 'text/html'});
    response.end(clientHtml);
}).listen(8080);
```

Now for the `socket.io` part (still in `server.js`):

```
var io = require('socket.io').listen(plainHttpServer);

io.set('origins', ['localhost:8080', '127.0.0.1:8080']);

io.sockets.on('connection', function (socket) {
  socket.on('message', function (msg) {
    if (msg === 'Hello') {
      socket.send('socket.io!');
    }
  });
});
```

That's the server, so let's make our `client.html` file:

```
<html>
<head>
</head>
<body>
<input id=msg><button id=send>Send</button>
<div id=output></div>

<script src="/socket.io/socket.io.js"></script>
<script>
(function () {
  var socket = io.connect('ws://localhost:8080'),
    output = document.getElementById('output'),
    send = document.getElementById('send');

  function logStr(eventStr, msg) {
    return '<div>' + eventStr + ':' + msg + '</div>';
  }
  socket.on('connect', function () {
    send.addEventListener('click', function () {
      var msg = document.getElementById('msg').value;
      socket.send(msg);
      output.innerHTML += logStr('Sent', msg);
    });
  });

  socket.on('message', function (msg) {
    output.innerHTML += logStr('Received', msg);
  });
})();
</script>
</body>
</html>
```

The final product is essentially the same as the previous recipe, except it will also work seamlessly in older browsers that aren't WebSocket compatible. We type Hello, press the **Send** button, and the server says **socket.io!** back.

How it works...

Instead of passing the HTTP server in a `options` object, we simply pass it to `listen` method.

We use `io.set` to define our origins whitelist and `socket.io` does the grunt work for us.

Next, we listen for the `connection` event on `io.sockets`, which provides us with a `socket` to the client (much like `request.accept` generates our `WebSocket` connection in the previous recipe).

Inside `connection`, we listen for the `message` event on `socket`, checking that the incoming msg is Hello. If it is we respond with `socket.io!`.

When `socket.io` is initialized, it begins to serve the client-side code over HTTP. So in our `client.html` file we load the `socket.io.js` client script from `/socket.io/socket.io.js`.

The client-side `socket.io.js` provides a global `io` object. By calling its `connect` method with our server's address, we acquire the relevant `socket`.

We send our Hello msg to the server, and say we have done so via the `#output` `div` element.

When the server receives Hello it replies `socket.io!`, which triggers our `message` event callback on the client side.

Now we have the `msg` parameter (different to our `msg` Hello variable) containing the message from the server, so we output it to our `#output` `div` element.

There's more...

`socket.io` builds upon the standard WebSocket API. Let's explore some of the additional functionality of `socket.io`.

Custom events

`socket.io` allows us to define our own events, other than `message`, `connect`, and `disconnect`. We listen to custom events after the same fashion (using `on`), but initiate them using the `emit` method.

Let's emit a custom event from the server to the client, then respond to the client by emitting another custom event back to the server.

We can use the same code as in our recipe, the only parts we'll change are the contents of the connection event listener callback in `server.js` (which we'll copy as `custom_events_server.js`) and the connect event handler in `client.html` (which we'll copy as `custom_events_client.html`).

So for our server code:

```
//require http, load client.html, create plainHttpServer
//require and initialize socket.io, set origin rules

io.sockets.on('connection', function (socket) {
  socket.emit('hello', 'socket.io!');
  socket.on('helloback', function (from) {
    console.log('Received a helloback from ' + from);
  });
});
```

Our server emits a `hello` event saying `socket.io!` to the newly connected client and listens out for a `helloback` event from the client.

So we modify the JavaScript in `custom_events_client.html` accordingly:

```
//html structure, #output div, script[src=/socket.io/socket.io.js] tag
socket.on('connect', function () {
  socket.on('hello', function (msg) {
    output.innerHTML += '<div>Hello ' + msg + '</div>';
    socket.emit('helloback', 'the client');
  });
});
```

When we receive a `hello` event, we log to our `#output div` (which will say `Hello socket.io!`) and emit a `helloback` event to the server, passing the client as the intended `from` parameter.

Namespaces

With `socket.io`, we can describe namespaces, or routes, and then access them as a URL through `io.connect` on the client:

```
io.connect('ws://localhost:8080/namespacehere');
```

A **namespace** allows us to create different scopes while sharing the same context. In `socket.io`, namespaces are used as a way to share a single WebSocket (or other transport) connection for multiple purposes. See <http://en.wikipedia.org/wiki/Namespace> and http://en.wikipedia.org/wiki/Computer_science.

Using a series of `i o.connect`, calls we are able to define multiple WebSocket routes. However, this won't create multiple connections to our server. `socket.io` multiplexes (or combines) them as one connection and manages the namespacing logic internally on the server, which is far less expensive.

We'll demonstrate namespacing by upgrading the code from the recipe *Transferring data between browser and server via AJAX* discussed in *Chapter 3, Working with Data Serialization*, to a `socket.io`-based app.

First, let's create a folder, call it `namespacing`, and copy the original `index.html`, `server.js`, `buildXml.js`, and `profiles.js` files into it. `Profiles.js` and `buildXml.js` are support files, so we can leave those alone.

We can strip down our `server.js` file, taking out everything to do with `routes` and `mimes` and reducing the `http.createServer` callback to its last `response.end` line. We no longer need the `path` module, so we'll remove that, and finally wrap our server in the `socket.io.listen` method:

```
var http = require('http');
var fs = require('fs');
var profiles = require('./profiles');
var buildXml = require('./buildXml');
var index = fs.readFileSync('index.html');
var io = require('socket.io').listen(
  http.createServer(function (request, response) {
    response.end(index);
  }).listen(8080)
);
```

To declare our namespaces with their connection handlers we use `io.of` as follows:

```
io.of('/json').on('connection', function (socket) {
  socket.on('profiles', function (cb) {
    cb(Object.keys(profiles));
  });

  socket.on('profile', function (profile) {
    socket.emit('profile', profiles[profile]);
  });
});

io.of('/xml').on('connection', function (socket) {
  socket.on('profile', function (profile) {
    socket.emit('profile', buildXml(profiles[profile]));
  });
});
```

In our `index.html` file we include `socket.io.js`, and connect to the namespaces:

```
<script src=socket.io/socket.io.js></script>
<script>
(function () { // open anonymous function to protect global scope
    var formats = {
        json: io.connect('ws://localhost:8080/json'),
        xml: io.connect('ws://localhost:8080/xml')
    };
    formats.json.on('connect', function () {
        $('#profiles').html('<option></option>');
        this.emit('profiles', function (profile_names) {
            $.each(profile_names, function (i, pname) {
                $('#profiles').append('<option>' + pname + '</option>');
            });
        });
    });

    $('#profiles, #formats').change(function () {
        var socket = formats[$('#formats').val()];
        socket.emit('profile', $('#profiles').val());
    });

    formats.json.on('profile', function(profile) {
        $('#raw').val(JSON.stringify(profile));
        $('#output').html('');
        $.each(profile, function (k, v) {
            $('#output').append('<b>' + k + '</b> : ' + v + '<br>');
        });
    });

    formats.xml.on('profile', function(profile) {
        $('#raw').val(profile);
        $('#output').html('');
        $.each($('#profile')[1].nextSibling.childNodes,
            function (k, v) {
                if (v && v.nodeType === 1) {
                    $('#output').append('<b>' + v.localName + '</b> : ' +
                        v.textContent + '<br>');
                }
            });
    });
})();

```

Once connected, the server emits a `profile` event with an array of `profile_names`, our client picks it up and processes it. Our client emits custom `profile` events to the relevant namespace, and each namespace socket listens for a `profile` event from the server, handling it according to its format (which is determined by namespace).

Namespaces allow us to separate our concerns, without having to use multiple `socket.io` clients (thanks to multiplexing). In similar fashion to the sub-protocol concept in WebSockets, we can restrict certain behaviors to certain namespaces giving us more readable code, and easing the mental complexity involved in a multifaceted real-time web app.

See also

- ▶ *Creating a WebSocket server* discussed in this chapter
- ▶ *Callbacks over socket.io transport* discussed in this chapter
- ▶ *Creating a real-time widget* discussed in this chapter

Callbacks over socket.io transport

With `socket.io` we can execute a callback function over WebSockets (or a relevant fallback). The function is defined client side, yet called server side (and vice versa). This can be a very powerful way to share processing resources and functionality between clients and servers.

In this recipe, we'll create a way for the server to call a client-side function that squares a number, and for the client to call a server-side function that sends Base64 encoding (`http://en.wikipedia.org/wiki/Base64`) of a sentence back to the client.

Getting ready

We simply need to create a new folder with `client.html` and `server.js` files.

How to do it...

On our server, as before, we load our `http` module and the `client.html` file, create our HTTP server, attach `socket.io`, and set the `origins` policy.

```
var http = require('http');
var clientHtml = require('fs').readFileSync('client.html');

var plainHttpServer = http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/html'});
  response.end(clientHtml);
}).listen(8080);

var io = require('socket.io').listen(plainHttpServer);
io.set('origins', ['localhost:8080', '127.0.0.1:8080']);
```

Next, in our connection event handler, we listen for the custom event give_me_a_number from the client, and emit a custom event give_me_a_sentence from the server.

```
io.sockets.on('connection', function (socket) {  
  
    socket.on('give me a number', function (cb) {  
        cb(4);  
    });  
  
    socket.emit('give me a sentence', function (sentence) {  
        socket.send(new Buffer(sentence).toString('base64'));  
    });  
  
});
```

In our client.html file:

```
<html>  
<head> </head>  
<body>  
<div id="output"></div>  
<script src="/socket.io/socket.io.js"></script>  
<script>  
    var socket = io.connect('http://localhost:8080'),  
        output = document.getElementById('output');  
  
    function square(num) {  
        output.innerHTML = "<div>" + num + " x " + num + " is "  
            + (num * num) + "</div>";  
    }  
  
    socket.on('connect', function () {  
        socket.emit('give me a number', square);  
  
        socket.on('give me a sentence', function (cb) {  
            cb('Ok, here is a sentence.');  
        });  
  
        socket.on('message', function (msg) {  
            output.innerHTML += '<div>Received: ' + msg + '</div>';  
        });  
    });  
  
</script>  
</body> </html>
```

How it works...

Immediately upon connection, both the server and client emit a custom `socket.io` event to each other.

For custom `socket.io` events, see the *There's more...* section of the previous recipe *Seamless fallbacking with socket.io*.

For both the client and server, when we pass a function as the second parameter of `emit`, `socket.io` creates a special parameter (`cb`) in the corresponding event listener's callback. `cb` is not, in this case, the actual function (if it was, it would simply run in the context from which it was called), but an internal `socket.io` function which passes the arguments back to the `emit` method on the other side of the wire. `emit` then passes these arguments into its callback, thus executing the function in local context.

We know that the functions run in their own context. If the server-side `give me a sentence` callback was executed on the client, it would fail because there is no `Buffer` object in browsers. If the `give me a number` ran on the server, it would fail since there is no DOM (Document Object Model) in Node (that is, there is no HTML, hence no `document` object and no `document.getElementById` method).

There's more...

`socket.io` can be a great basis for even higher-level specializing frameworks.

Shared functions with Nowjs

Nowjs extrapolates the `socket.io` callback feature into an even simpler API, allowing us to share functions via a global `now` object on the client and the `everyone.now` object on the server. Let's get the `now` module:

```
npm install now
```

Setting up Nowjs is eerily familiar:

```
var http = require('http');
var clientHtml = require('fs').readFileSync('now_client.html');

var plainHttpServer = http.createServer(function (request, response) {
  response.writeHead(200, {'Content-type': 'text/html'});
  response.end(clientHtml);
}).listen(8080);

var everyone = require('now').initialize(plainHttpServer);
everyone.set('origins', ['localhost:8080', '127.0.0.1:8080']);
```

The client `Html` file loads `now_client.html` instead, in place of `io` we have `everyone`, and rather than calling `listen` we call `initialize`. Everything else so far is the same (except, of course, requiring `now` instead of `socket.io`).

We'll re-implement our recipe using `now`, to finish off the server we put.

```
everyone.now.base64 = function(sentence, cb) {
  cb(new Buffer(sentence).toString('base64'));
}

everyone.on('connect', function() {
  this.now.square(4);
});
```

Let's save our server as `now_server.js`, and in `now_client.html` we write the following code:

```
<html>
<head></head>
<body>
<div id=put></div>
<script src="/nowjs/now.js"></script>
<script>
var put = document.getElementById('put');
now.square = function(num) {
  put.innerHTML = "<div>" + num + " x " + num + " is " + (num *
num) + "</div>";
}

now.ready(function() {
  now.base64('base64 me server side', function(msg) {
    put.innerHTML += '<div>Received: ' + msg + '</div>';
  });
});
</script>
</body></html>
```

NowJS makes function sharing trivial. On the server, we simply set up our `base64` method on `everyone.now`, which makes `base64` available to all clients.

Then we listen for the `connect` event, when it occurs we call `this.now.square`. In this context `this` is the socket to our client, so `this.now.square` calls the `now.square` method contained in `now_client.html`.

In our client, instead of loading `socket.io.js`, we include `now.js` whose route is exposed at server initialization. This provides us with the global `now` object, on which we set our `square` function method.

Once a connection is established (detected using `now.ready`), we call `now.base64` using a callback to pull the data from the server to the client.

See also

- ▶ *Seamless fallbacking with socket.io* discussed in this chapter
- ▶ *Creating a real time widget* discussed in this chapter
- ▶ *Browser-server transmission via AJAX* discussed in Chapter 3, *Working with Data Serialization*

Creating a real-time widget

The configuration options and well thought out methods of socket.io make for a highly versatile library. Let's explore the dexterity of `socket.io` by making a real-time widget that can be placed on any website and instantly interface with a remote `socket.io` server to begin providing a constantly updated total of all users currently on site. We'll name it the `Live Online Counter` (`Loc` for short).

Our widget is for easy user consumption and should require very little knowledge to get working, so we want a very simple interface. Loading our widget through a `script` tag, and then initializing the widget with a prefabricated `init` method would be ideal (this allows us to pre-define properties before initialization if necessary).

Getting ready

We'll need to create a new folder with some new files: `widget_server.js`, `widget_client.js`, `server.js`, and `index.html`.

Before we start let's also get the `socket.io-client` module from npm. We'll be using this to build our custom `socket.io` client-side code.

```
npm install socket.io-client
```

How to do it...

Let's create `index.html` to define the kind of interface we want as follows:

```
<html>
<head>
<style>
#loc {color: blue;} /* widget customization */
</style>
</head>
<body>
```

```
<h1> My Web Page </h1>
<script src=http://localhost:8081/loc/widget_server.js></script>
<script> locWidget.init(); </script>
</body></html>
```

We want to expose a route to /loc/widget_server.js which contains our loc widget. Behind the scenes, our widget will be held in widget_client.js.

So let's make it:

```
window.locWidget = {
  style: 'position: absolute; bottom: 0; right: 0; font-size: 3em',
  init: function () {
    var socket = io.connect('http://localhost:8081', {resource: 'loc'}),
        style = this.style;
    socket.on('connect', function () {
      var head = document.getElementsByTagName('head')[0],
          body = document.getElementsByTagName('body')[0],
          loc = document.getElementById('_lo_count');
      if (!loc) {
        head.innerHTML = '<style>#loc {' + style + '}</style>' +
          head.innerHTML;

        body.innerHTML +=
          '<div id=_loc>Online: <span id=_lo_count></span></div>';

        loc = document.getElementById('_lo_count');
      }
      socket.on('total', function (total) {
        loc.innerHTML = total;
      });
    });
  }
}
```

We need to test our widget from multiple domains, so we'll just implement a quick HTTP server (server.js) to serve index.html so we can access it by http://127.0.0.1:8080 and http://localhost:8080, this gives us our multiple domains.

```
var http = require('http');
var fs = require('fs');
var clientHtml = fs.readFileSync('index.html');

http.createServer(function (request, response) {
```

```
    response.writeHead(200, {'Content-type' : 'text/html'});
    response.end(clientHtml);
}).listen(8080);
```

Finally, the server for our widget, in `widget_server.js` we write:

```
var io = require('socket.io').listen(8081);
var sioclient = require('socket.io-client');
var widgetScript = require('fs').readFileSync('widget_client.js');
var url = require('url');

var totals = {};

io.configure(function () {
  io.set('resource', '/loc');
  io.enable('browser client gzip');
});

sioclient.builder(io.transports(), function (err, siojs) {
  if (!err) {
    io.static.add('/widget.js', function (path, callback) {
      callback(null, new Buffer(siojs + ';' + widgetScript));
    });
  }
});

io.sockets.on('connection', function (socket) {
  var origin = (socket.handshake.xdomain)
    ? url.parse(socket.handshake.headers.origin).hostname
    : 'local';

  totals[origin] = (totals[origin]) || 0;
  totals[origin] += 1;
  socket.join(origin);

  io.sockets.to(origin).emit('total', totals[origin]);
  socket.on('disconnect', function () {
    totals[origin] -= 1;
    io.sockets.to(origin).emit('total', totals[origin]);
  });
});
```

To test we need two terminals, in one we execute:

```
node widget_server.js
```

In the other we execute:

```
node server.js
```

If we point our browser to `http://localhost:8080`, open a new tab or window and navigate to `http://localhost:8080`. Again we will see the counter rise by one. If we close either window, it will drop by one. We can also navigate to `http://127.0.0.1:8080` to emulate a separate origin. The counter at this address is independent from the counter at `http://localhost:8080`.

How it works...

`widget_server.js` is the powerhouse of this recipe. We start by requiring `socket.io` and calling the `listen` method. Instead of passing it as an `httpServer` instance as in the prior tasks, we pass it the port number 8081. It helps if we think of `ws://localhost:8081` as a remote server that many client widgets connect to.

The next requirement is `socket.io-client`, which we load into our `sioclient` variable.

Through `sioclient` we can access the `sioclient.builder` method to generate a `socket.io.js` file. This concatenated to `widget Script` to effectively create a single JavaScript file containing `socket.io.js` and `widget_client.js`. We name the HTTP route to this file `widget.js`. When joining the `socket.io.js` file to our `widget Script`, we place a semicolon in between to ensure the scripts don't interfere with each other.

We pass two arguments to the `builder` method, the first is an array of transports. These are the various methods for creating the real-time effect (for example WebSockets, AJAX (xhr) polling). The earlier a transport appears in the array the more preferred it is. The array is generated with the `transports` method. Since we haven't set any transports during configuration, the default transports array is supplied. The second argument is a callback. Here we can grab the generated `socket.io.js` file via the `siojs` parameter.

Our widget is purely a JavaScript affair, for plugging in to any HTML page on any site. `socket.io` has an internal HTTP server used for supplying JavaScript client-side files. Instead of creating an HTTP server to provide our client widget code, we use `iostatic.add` (once we have our generated `socket.io.js`) to push a new route onto the internal HTTP server of `socket.io`. The second argument of `iostatic.add` is a callback function, which in turn has a function passed to it named `callback`.

`callback` is part of `socket.io`, it adds the content to the newly defined route. The first argument can point to a literal file, but we are dynamically generating the code, so we pass `null`. To the second parameter we pass `socket`'s `widget Script` into a `Buffer` and our route is created.

By altering the `resource` property to change the route to the internal HTTP server routes of `socket.io`, `io.set` helps us to brand our widget. As a result, our combined `widget.js` route won't appear at `/socket.io/widget.js` anymore, instead it will be at `/loc/widget.js`.

In order to connect to our configured static resource route from the client, we have to pass in an `options` object to `io.connect` in `widget_client.js`. Notice the absence of a slash prefix. A slash prefix is mandatory server side but for the client it is mandatorily omitted.

Now the stage is set for the actual socket action. We wait for a connection by listening for the `connection` event on `io.sockets`. Inside the event handler we use a few as yet undiscussed `socket.io` qualities.

A WebSocket is formed when a client initiates a handshake request over HTTP and the server responds affirmatively. `socket.handsake` contains the properties of the handshake.

`socket.handsake.xdomain` tells us whether the handshake was initiated from the same server. We will check for a cross-server handshake before retrieving the `hostname` of `socket.handsake.headers.origin`.

The origin of a same domain handshake is either `null` or `undefined` (depending on whether it's a local file handshake or localhost handshake). The latter causes `url.parse` to choke and the former isn't ideal. So for same domain handshakes we simply set our `origin` variable to `local`.

We extract (and simplify) the `origin` because it allows us to distinguish between websites that use `widget`, enabling site-specific counts.

To keep count, we use our `totals` object and add a property for every new `origin` with an initial value of 0. On each connection, we add 1 to `totals[origin]` listening to our `socket` for the `disconnect` event where we minus one from `totals[origin]`.

If these values were exclusively for server use, our solution would be complete. However, we need a way to communicate total connections to the client, but only for the site they are on.

`socket.io` has a handy new feature since `socket.io` version 7, which allows us to group sockets into rooms by using the `socket.join` method. We cause each socket to join a room named after its `origin`, then we use the `io.sockets.to(origin).emit` method to instruct `socket.io` to only `emit` to sockets belonging to the originating `sites` room.

In both the `io.sockets.connection` event and the `socket.disconnect` event, we `emit` our specific `totals` to corresponding sockets to update each client with the total amount of connections to the site the user is on.

`widget_client.js` simply creates a `div` called `#_loc` and updates it with any new totals it receives from `widget_server.js`.

There's more...

Let's look at how our app could be made more scalable as well as another use for WebSockets.

Preparing for scalability

If we were to serve thousands of websites, we would need scalable memory storage, and Redis would be a perfect fit. It operates in memory but also allows us to scale across multiple servers.



We'll need Redis installed, along with the `redis` module. For more information see *Chapter 4, Interfacing with Databases*.



We'll alter our `totals` variable so it contains a Redis client instead of a JavaScript object:

```
var totals = require('redis').createClient();
```

Now we modify our `connection` event handler like so:

```
io.sockets.on('connection', function (socket) {
  var origin = (socket.handshake.xdomain)
    ? url.parse(socket.handshake.headers.origin).hostname
    : 'local';
  socket.join(origin);

  totals.incr(origin, function (err, total) {
    io.sockets.to(origin).emit('total', total);
  });

  socket.on('disconnect', function () {
    totals.decr(origin, function (err, total) {
      io.sockets.to(origin).emit('total', total);
    });
  });
});
```

Instead of adding one to `totals[origin]`, we use the Redis INCR command to increment a Redis key named after `origin`. Redis automatically creates the key if it doesn't exist. When a client disconnects, we do the reverse and readjust `totals` using DECR.

WebSockets as a development tool

When developing a website, we often change something small in our editor, upload our file (if necessary), refresh the browser, and wait to see the results. What if the browser would refresh automatically whenever we saved any file relevant to our site? We can achieve this with `fs.watch` and WebSockets. `fs.watch` monitors a directory, executing a callback whenever a change to any files in the folder occurs (but it doesn't monitor sub-folders).



`fs.watch` is operating system dependent. To date, `fs.watch` has also been historically buggy (mostly under Mac OS X). Therefore until further advancements, `fs.watch` is suited purely to development environments rather than production (you can monitor how `fs.watch` is doing by viewing the open and closed issues here: <https://github.com/joyent/node/issues/search?q=fs.watch>).

Our development tool could be used alongside any framework, from PHP to static files. For a general server, let's take the recipe *Serving static files* from *Chapter 1, Making a Web Server*, to test our tool out on. We'll copy the files (including the `content` folder) from that recipe into a new folder, which we can name `watcher`.

For the server counterpart of our tool, we'll make `watcher.js`:

```
var fs = require('fs');
var io = require('socket.io').listen(8081);
var sioclient = require('socket.io-client');

var watcher = [';(function () {',
    '    var socket = io.connect('ws://localhost:8081');
    '    socket.on('update', function () {
        '        location.reload();
    '    });
    '})();'].join('');

sioclient.builder(io.transports(), function (err, siojs) {
    if (!err) {
        io.static.add('/watcher.js', function (path, callback) {
            callback(null, new Buffer(siojs + watcher));
        });
    }
});

fs.watch('content', function (e, f) {
    if (f[0] !== '.') {
        io.sockets.emit('update');
    }
});
```

Most of this code is familiar. We make a `socket.io` server (on a different port to avoid clashing), generate a concatenated `socket.io.js` plus client-side `watcher` code file, and add it to the static resources of `socket.io`. Since this is a quick tool for our own development uses, our client-side code is written as a string to the `watcher` variable.

The last piece of code calls the `fs.watch` method, where the callback receives the event name (`e`) and the filename (`f`).

We check that the filename isn't a hidden dotfile. During a save event some file systems or editors will change hidden files in the directory, thus triggering multiple callbacks sending several messages at high speed, which can cause issues for the browser.

To use it, we simply place it as a script within every page served (probably using server-side templating). However, for demonstration purposes, we simply place the following code into `content/index.html`:

```
<script src="http://localhost:8081/socket.io/watcher.js"></script>
```

Once we fire up `server.js` and `watcher.js`, we can point our browser to `http://localhost:8080` and see the familiar excited **Yay!** from Chapter 1, *Making a Web Server*. Any changes we make and save (either to `index.html`, `styles.css`, `script.js`, or the addition of new files) will be almost instantly reflected in the browser. The first change we can make is to rid ourselves of the alert box in `script.js` so changes can be seen fluidly.

See also

- ▶ *Creating a WebSocket server* discussed in this chapter
- ▶ *Seamless fallback with socket.io* discussed in this chapter
- ▶ *Storing and retrieving data with Redis* discussed in Chapter 4, *Interfacing with Databases*

6

Accelerating Development with Express

In this chapter we will cover:

- ▶ Generating Express scaffolding
- ▶ Defining and applying environments
- ▶ Dynamic routing
- ▶ Templating in Express
- ▶ CSS engines with Express
- ▶ Initializing and using a session
- ▶ Making an Express web app

Introduction

As excellent as Node's HTTP module is, Express repackages and streamlines its functionality to provide us with a fluid interface that makes for almost frictionless rapid web development.

In this chapter, we will progress from generating a vanilla Express project base to a fully-fledged Express web-application foundation with MongoDB providing backend data support.

Express 2 to Express 3



Throughout this chapter there are helpful boxes, like this one, that demonstrate how to migrate code from Express 2 to Express 3. The supporting code files contain both Express 2 and 3 code (with 3 commented out). Code files can be downloaded from <http://www.packtpub.com/support>.

Generating Express scaffolding

Express works both as a Node module and as a command-line executable. When we run `express` from the command line it generates a project skeleton for us, accelerating the preparation process.

Getting ready

We need to install `express` using the `-g` flag (install globally) in order to run the `express` executable from any directory.

```
sudo npm -g install express
```

We use `sudo` to ensure we get permission to install globally. This doesn't apply under Windows.

How to do it...

First, we decide upon the name of our app. Let's call it `nca` (Node Cookbook App) and simply do:

```
express nca
```

This will generate all of our project files under a new directory called `nca`. Before we can run our app, we must ensure that all dependencies are installed. We can find app dependencies in `nca/package.json`:

```
{
  "name": "application-name"
, "version": "0.0.1"
, "private": true
, "dependencies": {
    "express": "2.5.8"
, "jade": ">= 0.0.1"
}
}
```

For portability, it's important to have relevant modules installed within the `project` folder. To achieve this, we simply use the command line to `cd` into the `node_modules` directory and say:

```
npm install
```

This will make a new `node_modules` directory in our `project` folder, holding all dependencies.

How it works...

When we run the `express` executable, it creates a folder structure that's suited to Express development. In the project root, we have `app.js` and `package.json` files.

`package.json` is a convention established by the CommonJS group (a Javascript standards community), and has become the established method for describing modules and applications in Node.

The `npm install` command parses the dependencies from `package.json`, installing them locally in the `node_modules` folder.

This is significant because it ensures stability. Node's `require` function looks for a `node_modules` folder in the current working directory before searching parent directories. If we upgrade any modules in a parent directory, our project will continue to use the same version it was built upon. Installing modules locally allows us to distribute our project along with its dependencies.

The `app.js` file is our project boilerplate. We run our app with:

```
node app.js
```

The `express` executable adds three subdirectories to the project folder: `public`, `routes` and `views`.

`public` is the default folder that `app.js` passes to the `express.static` method, all our static files go here. It contains `images`, `javascripts`, and `stylesheets` folders each for their own self-evident purpose.

The `routes` folder holds `index.js` which is required by `app.js`. To define our routes, we push them onto Node's `exports` object (which we'll learn more about in *Chapter 9, Writing Your Own Node Modules*). Using `routes/index.js` helps to avoid clutter in `app.js`, and separates server code from route code. This way we can focus purely on our server, or purely on our routes.

Finally, `views` hold template files, which can really help with development acceleration. We'll be finding out how to work with views in the *Templating in Express* recipe.

There's more...

Let's take a few moments to go deeper into our generated project.

Picking apart app.js

Let's take a look at our generated app.js file:

```
var express = require('express')
, routes = require('./routes')
var app = module.exports = express.createServer();
// Configuration
app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});

app.configure('development', function(){
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
});

app.configure('production', function(){
  app.use(express.errorHandler());
});

// Routes

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

app.listen(3000, function(){
  console.log("Express server listening on port %d in %s mode", app.address().port, app.settings.env);
});
```

The app variable holds the result of `express.createServer`, which is essentially an enhanced `http.createServer`.

The `configure` method is invoked three times: once for global settings, once for production, and once for development. We'll be looking at production and development in greater detail in the next recipe.

Within the global `configure` callback, the default view directory (`views`) and engine (`jade`) are set, and the app is told to use `express.bodyParser`, `express.methodOverride`, `app.router`, and `express.static` middleware.

`bodyParser` made a brief appearance in *Chapter 2, Exploring the HTTP Object*, in the first recipe's *There's more...* section, in the form of `connect.bodyParser`.

Express includes all the standard Connect middleware, and is compatible with add-on Connect middleware. Therefore, in an Express project, `bodyParser` is loaded with `express.bodyParser.bodyParser` gives us access to any data sent from the client (like in a POST request).

`methodOverride` allows us to make pseudo DELETE and PUT requests from browser forms using a hidden input element called `_method`. For example:

```
<input type='hidden' name='_method' value='DELETE' >
```

There are many HTTP methods defined in the Hypertext Transfer Protocol documents (see <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>). However, browsers have typically only supported GET and POST, leaving other methods up to purpose built clients. Express works around lack of browser support using this hidden input to emulate a DELETE request while also supporting real DELETE requests from clients that do support the method.

`app.router` contains all the defined routes (anything passed to `app.get`, `app.post`, and so on). Routes are, in themselves, middleware. If `app.router` is not passed to `app.use`, the routes are automatically appended to the middleware stack. However, with manual inclusion we have the ability to place additional middleware after the `app.router` middleware.

Middleware is generally constructed as follows:

```
function (req, res, next) {  
  // do stuff  
  next();  
}
```

The `next` parameter is a sort of callback mechanism which loads any ensuing middleware. So when `app.router` is positioned above `express.static`, any dynamic routes accessed by a client will not unnecessarily trigger the static server to look for the non-existent file, unless those routes call `next` (alternatively, `next` can be called as a method of `req:req.next()`). For more information on middleware, see <http://www.expressjs.com/guide.html#middleware>.

Looking into routes/index.js

In `app.js`, `routes/index.js` is loaded with a `require`:

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

Note that `index` isn't specified, but if a directory is passed to `require`, Node will automatically look for `index.js`. Let's take a look:

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

Pushing `index` onto the `exports` object makes it available in `app.js` as `routes.index`, which is passed to `app.get` as follows:

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

The `routes.index` function should look familiar. It follows the pattern of an `http.createServer` callback, but is specific to the route. The request (`req`) and response (`res`) parameters are enhanced by Express. We'll be looking into these in coming recipes. The function itself simply calls the `res.render` method, which loads a template from `views/index.jade`, passing `title` as a variable which then outputs the generated content to the client.

See also

- ▶ *Defining and applying environments* discussed in this chapter
- ▶ *Dynamic routing* discussed in this chapter
- ▶ *Templating in Express* discussed in this chapter

Defining and applying environments

Development and production code have different requirements. For instance, during development we will most likely want a detailed error output to the client, for debugging purposes. In production, we protect ourselves from opportunistic exploitation by revealing as little internal information as possible.

Express caters to these differences with `app.configure` which allows us to define environments with specific settings.

Getting ready

We'll need our project folder (`nc-a`) from the previous recipe.

How to do it...

Let's take a look at the preconfigured environments:

```
app.configure('development', function(){
    app.use(express.errorHandler({ dumpExceptions: true, showStack: true
}));});  
  
app.configure('production', function(){
    app.use(express.errorHandler());
});
```

The generated file defines customized error-reporting levels for each environment. Let's add caching to our production server, which can be a hindrance in development.

We use `express.staticCache` to achieve this. However, it has to be called prior to `express.static`, so we move `express.static` from the global `configure` into both development and production environments, along with `staticCache` in production as follows:

```
app.configure(function() {
    app.set('views', __dirname + '/views');
    app.set('view engine', 'jade');
    app.use(express.bodyParser());
    app.use(express.methodOverride());
    app.use(app.router);
});  
  
app.configure('development', function(){
    app.use(express.static(__dirname + '/public'));
    app.use(express.errorHandler({ dumpExceptions: true, showStack: true
}));});  
  
app.configure('production', function(){
    app.use(express.staticCache());
    app.use(express.static(__dirname + '/public'));
    app.use(express.errorHandler({dumpExceptions: true}));
});
```

We also set `dumpExceptions` to `true` for the production `errorHandler`. This would allow us to quickly identify any problems that might occur once we've launched our app.

To use an environment, we set the special `NODE_ENV` variable on the command line as we are executing `node`:

```
NODE_ENV=production node app.js
```

Or on Windows:

```
set NODE_ENV=production  
node app.js
```

The development environment is default, so there's no need to use `NODE_ENV` to set it.

How it works...

Express provides a very convenient way for us to separate our work flow process. All we have to do is pass in the name of our environment with specific settings.

Under the hood, Express will be using `process.env` to determine the `NODE_ENV` variable, checking for a match against any defined environments. If `NODE_ENV` isn't set, Express defaults to loading the development environment.

There's more...

Let's look into some of the ways we can manage our environments.

Setting other environments

We could have other phases in our work flow that would benefit from specific settings. For instance, we may have a staging phase where we emulate as much of the production environment as possible on the development machine for testing purposes.

For example, if our production server requires us to run the process on a specific port (say port 80), that we cannot achieve on our development server (if we do not have root privileges for instance), we could add a staging environment and set a `port` variable that is only set to 80 in a production environment.

See *Chapter 10, Taking It Live*, for information on how to safely run Node on Port 80.

Let's add the staging environment underneath development as follows:

```
app.configure('development', function() {
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true
}));});  
  
// our extra staging environment
app.configure('staging', function() {
  app.use(express.errorHandler({dumpExceptions: true}));});
```

Now we'll add the port logic as shown in the following code:

```
var port;
app.configure('production', function(){
  port = 80;
  app.use(express.errorHandler({dumpExceptions: true}));
});

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

app.listen(port || 3000);
```

So our port is set based upon the environment, if port is empty we default to 3000.

We could initialize our production server with:

```
sudo NODE_ENV=production node app.js
```

Or for Windows:

```
set NODE_ENV=production
node app.js
```

When attempting to run the server with NODE_ENV set to production, if we receive `TypeError: Cannot read property 'port' of null`, it's likely that a service is already running on port 80. We would need to stop this service in order to test our code. For instance, if Apache is running on our system, it's probably hosting through port 80. We can stop Apache with `sudo apachectl -k stop` (or `net stop apache2.2` on Windows).

Changing NODE_ENV permanently

If we are in a staging process, we may not wish to type `NODE_ENV=staging` every time we load our app. The same applies to production. While the server would be started a lot less, we would have to remember to set `NODE_ENV` when restarting.

We can make things easier on Unix-type systems (Linux or Max OS X), with the `export` shell command as follows:

```
export NODE_ENV=staging
```

This only sets `NODE_ENV` while our terminal is open. To make it permanent, we add this line to our home directory's `rc` file. The `rc` file is named depending upon the shell. For bash, it's located in `~/.bashrc` (where `~` is the home folder). Other shells, such as `sh` and `ksh`, would be `~/.shrc`, `~/.kshrc`, and so on.

To permanently set NODE_ENV, we can use:

```
echo -e "export NODE_ENV=staging\n" >> ~/.bashrc
```

Where staging is our desired environment and bash is our shell.

In Windows, we use set and setx:

```
set NODE_ENV=staging  
setx NODE_ENV=staging
```

set takes immediate effect, but is lost once the command prompt is closed. setx applies permanently but not until we open a new command prompt, so we use both.

See also

- ▶ *Generating Express scaffolding* discussed in this chapter
- ▶ *Deploying to a server environment* discussed in *Chapter 10, Taking It Live*
- ▶ *Making an Express web app* discussed in this chapter
- ▶ *Initializing and using a session* discussed in this chapter

Dynamic routing

In the very first recipe of this cookbook, *Setting up a router*, we explored various ways to set up routing in Node. Express provides a far superior and very powerful routing interface which we'll explore in this recipe.

Getting ready

We'll be working with our nca folder.

How to do it...

Let's say we want to add a page for a fictional character by the name of Mr Page. We'll name the route page, so in the routes section of app.js we add the following code:

```
app.get('/page', function (req, res) {  
  res.send('Hello I am Mr Page');  
});
```

We can also define flexible routes, and grab the requested route using req.params, like so:

```
app.get('/:page', function (req, res) {  
  res.send('Welcome to the ' + req.params.page + ' page');  
});
```

It's okay to throw our callbacks directly into `app.get` while developing, but in the interest of a clutter-free `app.js` let's take our callbacks and load them from `routes/index.js` as follows:

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

exports.mrpage = function (req, res) {
  res.send('Hello I am Mr Page');
};

exports.anypage = function (req, res) {
  res.send('Welcome to the ' + req.params.page + ' page');
};
```

And back in our `app.js` file our routes become:

```
// Routes
app.get('/', routes.index);
app.get('/page', routes.mrpage);
app.get('/:page', routes.anypage);
```

How it works...

We create the `/page` route using `app.get`. Then outline how we wish to respond to that route in the callback of `app.get`. In our example, we use `res.send` (an enhanced `res.write`) to output simple text. This is our inflexible dynamic route.

Express also provides flexible route capabilities using placeholders. In the main recipe, we defined a `:page` placeholder. When the placeholder is filled in by a request (for example, `/anyPage YouLi ke`), the fulfillment of the placeholder is added to `req.params` according to its name. So in this case `req.params.page` would hold `/anyPage YouLi ke`.

When a user loads `localhost:3000/page` they see **Hello I am Mr Page**, when they access `localhost:3000/absolutelyAnythingElse` they get **Welcome to the absolutelyAnythingElse page**.

There's more...

What other things can we do with Express routes?

Route validation

We can restrict flexible routes to specific character ranges using pieces of Regular Expression syntax, like so:

```
app.get('/:page([a-zA-Z]+)', routes.anypage);
```

We pass a character match, [a - z A - Z] along with a plus (+). This will match the characters one or more times. As a result, we limit our : page parameter to letters only.

Therefore, `http://localhost:3000/moo` will give **Welcome to the moo page**, whereas `http://localhost:3000/moo1` will return a 404 error.

Optional routes

We can also define optional routes using the question mark (?):

```
app.get('/:page/:admin?', routes.anypageAdmin);
```

We would place this in our `app.js` file, underneath our other defined routes.

Our `anypageAdmin` function in `routes/index.js` could go like this:

```
exports.anypageAdmin = function(req, res) {
  var admin = req.params.admin
  if (admin) {
    if(['add','delete'].indexOf(admin) !== -1) {
      res.send('So you want to ' + req.params.admin + ' ' + req.params.page + '?');
      return;
    }
    res.send(404);
  }
}
```

We check for the existence of the : admin placeholder. If a route fulfills it, we verify that it is allowed (either add or delete) and send a tailored response. If the route is not allowed, we send a 404 error.

While the query wildcard (?) can be appropriate for lots of similar routes, if we only had our `add` and `delete` routes and there was no possibility of adding more routes later, we could implement this functionality in a much cleaner way.

In `app.js` we could put:

```
app.get('/:page/:admin((add|delete))', routes.anypageAdmin);
```

And in `index/routes.js`:

```
exports.anypageAdmin = function (req, res) {
  res.send('So you want to ' + req.params.admin + ' ' + req.params.page + '?');
}
```

Asterisks wildcards

We can use the asterisks (*) as a wildcard for general matching requirements. For instance, let's add the following route:

```
app.get('/:page/*', routes.anypage);
```

And change `routes.anypage` to the following:

```
exports.anypage = function (req, res) {

  var subpage = req.params[0],
    parentPage = subpage ? ' of the ' + req.params.page + ' page' :
  '';

  res.send('Welcome to the ' +
    (subpage || req.params.page) + ' page' + parentPage);

};
```

Now if we access `localhost:3000/foo/bar` we get **Welcome to the bar page of the foo page**, but if we just access `localhost:3000/foo` we see **Welcome to the foo page**.

We could also get a little wild and apply this to Mr Page's route as follows:

```
app.get('/ *page*', routes.mrpage);
```

Now any route containing the word `page` will get a message from **Mr Page**.

See also

- ▶ *Setting up a router* discussed in *Chapter 1, Making a Web Server*
- ▶ *Making an Express web app* discussed in this chapter
- ▶ *Templating in Express* discussed in this chapter

Templating in Express

A fundamental part of the Express framework is its use of views. A view is simply a file that holds template code. Express helps us to separate our code into operationally distinct concerns. We have server code in `app.js`, route-specific functionality in `routes/index.js`, and then we have our output generating logic in the `views` folder. A template language provides a basis for defining dynamic logic-driven content, and the template (or view) engine converts our logic into the final HTML which is served to the user. In this recipe, we'll use Express' default view engine, Jade, to process and present some data.

In the *There's more...* section, we'll find out how to change the view engine.

A list of supported template engines can be found at <https://www.github.com/visionmedia/express/wiki>. Comparisons of various template engines can be found at <http://paularmstrong.github.com/node-templates/>.

Getting ready

For our data, we'll be using the `profiles.js` object we created back in the first recipe of *Chapter 3, Working with Data Serialization*. We'll need to copy it into the root of our `node` folder.

How to do it...

Let's keep it simple and strip any routes we've added to `app.js`. We just want our top-level route.

Since Jade is set as the default view engine in `app.configure`, there's nothing else we need to do with `app.js` in this example.

In `routes/index.js`, we'll strip all routes except for `index`.

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

The `res.render` method loads the Jade template in `views/index.jade`. We're going to use `index.jade` as a view for our `profiles.js` object data, so we need to make it available to our `index` view.

We do this by passing it through the `options` object of `res.render`:

```
var profiles = require('../profiles.js');
exports.index = function(req, res){
  res.render('index', { title: 'Profiles', profiles: profiles })
};
```

Notice we also changed the `title` property to 'Profiles'.

All we do now is edit `views/index.jade`. The generated `index.jade` contains the following:

```
h1= title
p Welcome to #{title}
```

We're going to add a table to the page that outputs the details of each person in the `profiles.js` object:

```
table#profiles
  tr
    th Name
    th Irc
    th Twitter
    th Github
    th Location
    th Description
    each profile, id in profiles
      tr(id=id)
        each val in profile
          td #{val}
```

To test we start our app:

```
node app.js
```

And then navigate to `http://localhost:3000` to see something like the following:

Profiles					
Welcome to Profiles					
Name	Irc	Twitter	Github	Location	Description
Ryan Dahl	ryah	ryah	ry	San Francisco, USA	Creator of node.js
Isaac Schlueter	isaacs	izs	isaacs	San Francisco, USA	Author of npm, core contributor
Bert Belder	piscisaureus	piscisaureus	piscisaureus	Netherlands	Windows support, overall contributor
TJ Holowaychuk	tjholowaychuk	tjholowaychuk	visionmedia	Victoria, BC, Canada	Author of express, jade and other popular modules
Felix Geisendorfer	felixge	felixge	felixge	Berlin, Germany	Author of formidable, active core developer

How it works...

`res.render` pulls `index.jade` from the `views` folder, even though the first parameter is simply `index`. Express knows that a Jade file inside the `views` directory is intended because `app.configure` of `app.js` contains the following code:

```
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
```

The second parameter is an object, holding two properties: `title` and `profiles`. These object properties become local variables within the Jade view. We output the variables either by return value buffering with a preceding equals (=) sign, or by using Jade's interpolation, wrapping it like so: `#{title}`.

Jade is a lean templating language. It uses bracket-stripped markup tags and has an indentation-based syntax with an alternative block expansion option (where we use the colon instead of an indent to signify nesting). It also has a minimal syntax set for defining `id` and `class` attributes using the hash (#) and dot (.) respectively.

For instance, the following Jade:

```
table#profiles
  th Name
```

Would create the following HTML:

```
<table id=profiles><th>Name</th></table>
```



To learn more about the Jade language, check out its GitHub page:
<https://www.github.com/visionmedia/jade>.



Jade also processes iteration logic. We used two `each` Jade iterators to pull the values from our `profiles` object as follows:

```
each profile, id in profiles
  tr(id=id)
    each val in profile
      td #{val}
```

This code traverses the `profiles` object, loading each ID (`ryan`, `isaac`, `bert`, and so on) into a new `id` variable, and each object containing profile information into a `profile` object variable.

Underneath our first `each`, we indent `tr(id=id)`. Unlike JavaScript, indentation in Jade is part of the logic, so getting it right is essential.

This tells Jade that for each profile we want to output a `<tr>` tag with the `id` attribute set to the ID of the profile. We don't use the hash (#) shorthand to set the `id` attribute in this case since we need Jade to evaluate our `id` variable. `tr#id` would generate `<tr id=id>` for each profile, whereas `tr(id=id)` generates `<tr id=ryan>` or `isaac`, or `bert` and so forth.

Underneath `tr` we indent again, indicating that whatever comes next should be nested within the `<tr>` tags. Again we use `each` to traverse the values of each sub-object, indenting beneath with a `td` that holds each value of the profile.

There's more...

Let's take a look at some of the other templating capabilities and features Express has to offer.

Using other template engines

Express supports a variety of alternative template engines, unsupported engines can be adapted to Express without excessive hassle.

The `express` executable will only generate Jade or EJS-based project scaffolding. To generate EJS we simply pass `ejs` to the `-t` flag:

```
express -t ejs nca
```

Instead of generating an Express project with EJS as the default view engine, let's convert our existing project (we'll start by copying it to `nca_ejs`).

First, we need to edit dependencies in `package.json`:

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "2.5.8",
    "ejs": "0.7.1"
  }
}
```

We've simply removed `jade` and put `ejs` in its place. Now we do:

```
npm install
```

So npm will put the EJS module into the `node_modules` folder.

Finally, we change our view engine in `app.configure` as follows:

```
app.configure(function() {
  app.set('views', __dirname + '/views');
  app.set('view engine', 'ejs');
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
});
```

This technique will work for any Express-supported template engine. There's no need to require the EJS module, Express takes care of that behind the scenes.

EJS templates

Since we've set up `nca_ejs`, we may as well go ahead and rewrite our index view in Embedded JavaScript.

In `nca_ejs/views` we add a new file, `index.ejs`, and put:

```
<h1> <%= title %></h1>
<p> Welcome to <%= title %></p>

<table>
<tr><th>Name</th><th>IRC</th><th>Twitter</th>
<th>GitHub</th><th>Location</th><th>Description</th></tr>

<% Object.keys(profiles).forEach(function(id) {%
  <tr>
    <% Object.keys(profiles[id]).forEach(function(val) { %
      <td><%= profiles[id][val]; %></td>
    <% }); %>
  </tr>
<% }); %>
</table>
```

`<% and %>` denote embedded JavaScript. If JavaScript happens to wrap any HTML code, the HTML is processed as if it's part of the JavaScript. For instance, in our `forEach` callbacks we have `<tr>` and `<td>`, these are included as output from each loop.

When the opening tag is accompanied by an equals sign (`<%=`), it evaluates any given JavaScript variable and pulls it into the generated output. For example, in our first `<h1>` we output the `title` variable.

Literal JavaScript in Jade

Jade can also process plain JavaScript. Let's use that to our advantage to output our table headers in a more concise, dry fashion:

```
- var headers = ['Name', 'Irc', 'Twitter', 'Github', 'Location',
'Description'];
table#profiles
  tr
    each header in headers
      th= header
    each profile, id in profiles
      tr(id=id)
        each val in profile
          td #{val}
```

A dash (-) at the beginning of a line informs Jade that we're using plain JavaScript. Here we simply create a new array called `headers` and then use Jade's `each` iterator to output our headers, using the equals (=) sign to evaluate the `header` variable.

We could alternatively create our array in Jade as follows:

```
headers = ['Name', 'Irc', 'Twitter', 'Github', 'Location',
'Description'];
```

Jade then compiles this to the embedded JavaScript in the preceding example, including the `var` declarative.

Jade partials

Partials are described as mini-views or document-fragments. They are mainly intended for automatic templated iteration over an array (a collection), although they'll also work with objects.

For instance, instead of saying:

```
tr(id=id)
  each val in profile
    td #{val}
```

We can create a view file, which we'll call `row.jade`, in which we write:

```
td= row
```

Back in `index.jade`, we replace our `each` iterator with `partial` as follows:

```
each profile, id in profiles
  tr(id=id)
    != partial('row', {collection: profile})
```

`!=` tells Jade to not only buffer what `partial` returns, but also to refrain from escaping the returned HTML. If we don't include the exclamation mark Jade replaces HTML characters with their special entity codes, (for example, `<` becomes `<`).

We pass `'row'` into `partial` which tells Jade to use the `row.jade` view as the partial. We pass an object with a `collection` property as the next parameter. If our profile was a simple array, we could simply pass the array and Jade would generate a `td` tag for each value in the array. However, the `profile` variables are objects, so passing it to `collection` causes Jade to traverse the values as if they were a simple array.

Each value in our `collection` (`Ryan Dahl`, `ryah`, `Creator of Node.js`, and so on), is referenced by the name of the view. So in our `row.jade` view, we use the `row` variable to grab each value. We can customize it by using the `as` property as follows:

```
!= partial('row', {collection: profile, as: 'line'})
```

Then in `row.jade` we would change `row` to `line`:

```
td= line
```

Express 2 to Express 3

In an effort to simplify viewing system internals and make it easier for template engines to integrate with Express, version 3 will no longer support partials. In Express 3, instead of a `row.jade` file, and using the call to `partial`, we could instead say:



```
each profile, id in profiles
  tr(id=id)
    each row in profile
      td= row
```

Express partials

One of the excellent things about partials is we can use them in our Express routes on the `response(res)` object. This is particularly remarkable because it allows us to seamlessly send fragments of HTML to AJAX or WebSocket requests, all while generating content from the same fragments (inside our views) for whole-page requests.

At the end of `index.jade` (the partials version), we'll insert a small proof of concept script:

```
script(src='http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js')
script
  set Timeout(function () {
```

```
$.get('/', function (row) {
  $('#profiles tbody').append(row);
}),
}, 1500);
```

This will wait for one and a half seconds and then make an AJAX request to our `index` route. So let's modify our `index` route in `routes/index.js`:

```
var profiles = require('../profiles.js');

exports.index = function(req, res){
  if (req.xhr) {
    res.partial('row', {collection: profiles.ryan});
  }
  res.render('index', { title: 'Profiles', profiles: profiles});
};
```

If the request is an XMLHttpRequest (AJAX), we generate a new table row out of the `ryan` profile.

Now when we load `http://localhost:3000`, after a short delay Ryan's profile appears at the bottom of the table.

Express 2 to Express 3

Express 3 doesn't support partials, (neither in template logic nor in app code), so we would have to approach this a different way. For instance, we could send a JSON representation of the profiles and have the browser loop through it to populate the table.

As of this time of writing, there is no replacement middleware for partials, but there may well be in the near future.

Jade includes

Includes help us to separate and re-use pieces of template code. Let's put our `profiles` table into its own view. We'll call it `profiles.jade`.

To include `profiles.jade` from the `index.jade` file we simply do the following:

```
h1= title
p Welcome to #{title}
include profiles
```

layout.jade

Express 2 to Express 3



Layouts are also being axed from Express 3 in favor of block inheritance. So instead of any rendered views being implicitly wrapped up into the body variable and rendered within `layout.jade`, we now have to explicitly declare a block and then insert that block into our body.

Also included in a generated project is the `layout.jade` view. This is a special view that is intertwined with Express logic. Any rendered views are packaged into a `body` variable, which is then passed into `layout.jade`. So in our case, we tell `res.render` to assemble `index.jade`. Express converts `index.jade` to HTML, and then internally renders `layout.jade`, passing the generated HTML in a `body` variable. `layout.jade` allows us to head and foot our views. To disable this feature for the entire app, we use `app.set('view options', {layout: false})`. To prevent it from applying to a particular render, we simply pass `layout: false` to the options object of `res.render`.

Express 2 to Express 3



So in `layout.jade`, instead of `body!=body` we have the following:

```
body
block content
```

And at the top of `index.jade` we would inherit from `layout.jade` using `extends layout` and then define the `content` block, which would be loaded into the body of `layout.jade`:

```
extends layout
block content
h1= title
p Welcome to #{title}
/- the rest of our template...
```



All Jade Express code examples have an additional folder named `views - Express 3` containing equivalent templates, which follow the explicit block inheritance pattern instead of implicit layout wrapping.

See also

- ▶ *CSS Engines with Express* discussed in this chapter
- ▶ *Making an Express web app* discussed in this chapter
- ▶ *Generating Express scaffolding* discussed in this chapter

CSS engines with Express

Once we have our HTML, we'll want to style it. We could of course use raw CSS, but Express integrates nicely with some select CSS engines.

Stylus is one such engine. It's written with Express in mind, and as a syntax it follows many of the design principles found in Jade.

In this recipe, we're going to put Stylus in the spotlight, learning how we can use it to apply styles to our `profiles` table from the previous recipe.

Getting ready

We'll need our `node_modules` folder as it was left in the previous recipe.

How to do it...

First, we need to set up our app to use Stylus.

If we were starting a new project, we could use the `express` executable to generate a Stylus-based Express project, as follows:

```
express -c stylus ourNewAppName
```

This would generate a project where `stylus` is a dependency in `package.json`, with an extra line in `app.js` within `app.configure`:

```
app.use(require('stylus').middleware({ src: __dirname + '/public'}));
```

However, since we've already got a project on the hotplate, let's modify our existing app to use Stylus.

In `package.json`:

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "2.5.8",
    "jade": ">= 0.0.1",
    "stylus": "0.27.x"
  }
}
```

Then on the command line run the following command:

```
npm install
```

Finally in `app.js`, inside `app.configure` we insert the following code:

```
app.use(require('stylus').middleware({
  src: __dirname + '/views',
  dest: __dirname + '/public'
}));
```

Notice we've set a different `src` and added a `dest` property to the generated code.

We're going to put our Stylus files in `views/stylesheets`. So let's make that directory and place a new file in it, which we'll call `style.styl`. Express will find this file, placing generated CSS in the corresponding folder (`stylesheets`) of the `public` directory.

To start our Stylus file, we'll copy the current CSS from `stylesheets/style.css` as follows:

```
body {
  padding: 50px;
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;
}
a {
  color: #00b7ff;
}
```

Stylus is fully compatible with plain CSS, but for learning purposes let's convert it into the minimal indentation-based format:

```
body
  padding 50px
  font 14px "Lucida Grande", Helvetica, Arial, sans-serif;
a
  color #00B7FF
```

Now we'll style our `#profiles` table from the previous recipe.

We can apply consistent padding to our `td` and `th` tags as well as our `#profile` table utilizing Stylus' `@extend` directive as follows:

```
.pad
  padding 0.5em
#profiles
  @extend .pad
  th
    @extend .pad
  td
    @extend .pad
```

As new CSS properties are introduced into browsers, they often come with vendor-specific prefixes until the implementation is considered mature and stable. One such property is `border-radius`, on Mozilla browsers it's `-moz-border-radius`, on WebKit types it's referenced as `-webkit-border-radius`.

Writing and maintaining this sort of CSS can be quite involved, so let's use a Stylus mixin to make our lives easier:

```
borderIt(rad = 0, size = 1px, type = solid, col = #000)
  border size type col
  if rad
    -webkit-border-radius rad
    -moz-border-radius rad
    border-radius rad
```

Now, we'll apply our mixin to the `#profiles` table and all the `td` elements:

```
#profiles
  borderIt 20px 2px
  @extend .pad
  th
    @extend .pad
  td
    @extend .pad
    borderIt(col: #000 + 80%)
```

So our `#profiles` table now looks as shown in the following screenshot:

Profiles					
Welcome to Profiles					
Name	Irc	Twitter	Github	Location	Description
Ryan Dahl	ryah	ryah	ry	San Francisco, USA	Creator of node.js
Isaac Schlueter	isaacs	izs	isaacs	San Francisco, USA	Author of npm, core contributor
Bert Belder	piscisaureus	piscisaureus	piscisaureus	Netherlands	Windows support, overall contributor
TJ Holowaychuk	tjholowaychuk	tjholowaychuk	visionmedia	Victoria, BC, Canada	Author of express, jade and other popular modules
Felix Geisendorfer	felixge	felixge	felixge	Berlin, Germany	Author of formidable, active core developer

How it works...

As a module, `stylus` can operate independent of Express. However, it also has a convenient `middleware` method which can be passed into `app.use`.

When the `express` executable generates a Stylus-powered project, only the `src` property is set, which means Stylus pulls files with a `.styl` extension and converts them to `.css` files in the same folder. When we set `dest`, we load our Stylus code from one place and save it in another.

Our `src` is `views` and `dest` is `public`, but even though we put our `styles.styl` in a subdirectory of `views`, Stylus still finds it and places it in the corresponding subdirectory of the `dest` folder.

The `layout.jade` file includes a `link` tag to `/stylesheets/style.css`. So when we created `style.styl` in `views/stylesheets`, the generated CSS was written to `public/stylesheets`. Since our static server directory is set to `public`, requests for `/stylesheets/style.css` are served from `public/stylesheets/style.css`.

We used several Stylus features to create our stylesheet.

The `@extend` directive is based upon the concept of inheritance. We make a class and then use `@extend` to apply all the qualities of that class to another element. Our use of `@extend` in the recipe creates the following CSS:

```
.pad,  
#profiles,  
#profiles th,  
#profiles td { padding: 0.5em }
```

The larger our styles base becomes the more the `@extend` directive tends to ease maintenance and readability.

We make it easier to define a border, with rounded corners if desired, by using a mixin. Stylus mixins allow us to define default values as we set the parameters. If we mixed in `borderIt` with no arguments, it would generate a 1 pixel-wide, right-angled solid black border according to its defaults.

We first use `borderIt` on the `#profiles` table, passing in `20px` and `2px`. There's no need to use parentheses—Stylus understands it's a mixin. The first parameter (`20px`) in our mixin is named `rad`. Since `rad` has specified the `borderIt`, mixin goes ahead and outputs the various vendor prefixes along with the desired radius. The second parameter overwrites our `border-width` default.

We do need parentheses when we apply `borderIt` to the `td` elements, because we define our options using a `kwarg` (a keyword argument). All we want to do is set the color, so instead of supplying all preceding parameters, we simply reference the desired parameter as a property. The color we pass is `#000 + 80%`. This is not valid CSS but Stylus understands.

There's more...

Let's explore some more Stylus features, and also find out how to use the alternative CSS engine, LESS, as Express middleware.

Nested mixins and rest parameters

Let's take a look at reusing mixins in other mixins, and Stylus' rest parameter syntax (essentially a single parameter that consumes any following parameters, compiling them into an array).

We could soften the edges of our table further by rounding the relevant angles of the corner `<t d>` elements, such that they match the rounded nature of the outer border.

We need to be able to set a radius for an individual corner. Vendor implementations differ on their approach to this. In Mozilla-based browsers, the corner is defined after radius with no dash, for example:

```
- moz-border-radius-topleft: 9px
```

Whereas WebKit conforms to the specification (except the prefix) with the following code:

```
-webkit-border-top-left-radius
```

Let's create another mixin dedicated to creating the rounded corners CSS, whether corners are equal or not.

```
rndCorner(rad, sides...)
  if length(sides) is 2
    - moz-border-radius-{{sides[0]}} {{sides[1]}} rad
    - webkit-border-{{sides[0]}}-{{sides[1]}}-radius rad
    border-{{sides[0]}}-{{sides[1]}}-radius rad
  else
    - webkit-border-radius rad
    - moz-border-radius rad
    border-radius rad
```

`sides` is a rest parameter. It swallows up all remaining arguments. We need two sides for a corner, for example, top left. So we use a conditional statement to check if length of the remaining arguments is 2 (instead of `is` we could have used `==`).

If we have our sides, we integrate them into the various browser-specific CSS. Notice when including variables in a property we escape them with curly brackets (`{}`). If sides aren't specified we set the radius to all sides, as in our recipe.

Now we can call this mixin from our borderIt mixin as follows:

```
borderIt(rad = 0, size = 1px, type = solid, col = #000)
  border size type col
  if rad { rndCorner(rad) }
```

We didn't have to wrap the conditional statement with braces. This just allows us to keep our if statement and mixin call on the same line. It's the equivalent to the following code:

```
borderIt(rad = 0, size = 1px, type = solid, col = #000)
  border size type col
  if rad
    rndCorner(rad)
```

Finally, we apply our single corners to the relevant td elements:

```
tdRad = 9px
"profiles"
  borderIt 20px 2px
  @extend .cell
  th
    @extend .cell
  td
    @extend .cell
    borderIt(col: #000 + 80%)
  tr
    & nth-child(2)
      td:first-child
        rndCorner tdRad top left
      td:last-child
        rndCorner tdRad top right
    & last-child
      td:first-child
        rndCorner tdRad bottom left
      td:last-child
        rndCorner tdRad bottom right
```

Our first borderIt now calls the rndCorner mixin inferentially because it sets a radius. The second borderIt won't call rndCorner, which is great because we want to call it ourselves on specific elements.

We use the special ampersand (&) reference to cite the parent tr element. We use CSS's :nth-child(2) to select the second row of our table. The first row consists of th elements. The same applies for first-child and last-child, which we use to apply the appropriate corners to our td elements.

While this `:nth-child` and `:last-child` pseudo-selectors won't work in Internet Explorer 8 and below, neither will `border-radius`, so this is one of the few cases we can use it and still be cross-browser compatible, progressively enhancing in more modern browsers.

Playing with colors

Stylus does some amazing things with color. It has functions that allow us to lighten/darken, (de)saturate, hue adjust, and even mix colors together.

Let's color our table in:

```
#profiles
  borderIt 20px, 2px
  @extend .pad
  background: #000;
  color: #fff;
  th
    @extend .pad
  td
    @extend .pad
    background blue + 35%
    borderIt(col: @background)
    color pink - green - brown + gold - green
    color desaturate(@color + 100, 100)
    & hover
      color @background + 180deg
      background desaturate(@background, 40)
      border-color @background
```

We can reference values of any properties already set for an element. We use the `@background` property lookup variable consistently throughout this piece of code, but in many cases it holds a different value.

To start off, we invert our `#profile` table, setting `color` to white, and `background` to black. We next apply color to our `td` elements, obtaining a lighter shade of blue by adding 35% to it. We match our `td` borders to their `background` colors with the `@background` property lookup.

Then we just go wild with color mixing, eventually setting the text color of our `td` to a color not far from the original pink. We then pass `@color` to `desaturate` while also lightening it using `+`. Next, we set the hover text color by adding 180 degrees to our `@background` color, obtaining the complimentary hue. We also `desaturate` our `background` and match `border-color` (`@background` now matches the `desaturated` `background`, whereas when we set the color on hover it matched the pre-hover `background` color).

So now our table looks as shown in the following screenshot:

Profiles					
Welcome to Profiles					
Name	Irc	Twitter	Github	Location	Description
Ryan Dahl	ryah	ryah	ry	San Francisco, USA	Creator of node.js
Isaac Schlueter	isaacs	izs	isaacs	San Francisco, USA	Author of npm, core contributor
Bert Belder	piscisaureus	piscisaureus	piscisaureus	Netherlands	Windows support, overall contributor
TJ Holowaychuk	tjholowaychuk	tjholowaychuk	visionmedia	Victoria, BC, Canada	Author of express, jade and other popular modules
Felix Geisendorfer	felixge	felixge	felixge	Berlin, Germany	Author of formidable, active core developer

Using the LESS engine

LESS may be suitable as a more familiar and verbose alternative to Stylus. We can use LESS with Express by replacing:

```
app.use(require('stylus').middleware({
  src: __dirname + '/views',
  dest: __dirname + '/public'
}));
```

With:

```
app.use(express.compiler({
  src: __dirname + '/views',
  dest: __dirname + '/public',
  enable: ['less']
}));
```

To ensure this works, we should also change our `package.json` file as follows:

```
{
  "name": "application-name"
, "version": "0.0.1"
, "private": true
, "dependencies": {
    "express": "2.5.8"
, "jade": ">= 0.0.1"
, "less": "1.3.x"
}
}
```

And run the following command:

```
npm install
```

To test it out, we'll rewrite our recipe in LESS.

Some Stylus features have no equivalent in LESS. Instead of using `@extend` to inherit our `.pad` class, we'll convert it into a mixin. There are no `if` conditionals in LESS either, so we'll declare the `.borderIt` mixin twice, the second time using the `when` statement.

```
body { padding: 50px;
        font: 14px "Lucida Grande", Helvetica, Arial, sans-serif; }

a { color: #00B7FF; }

.pad() { padding: 0.5em }

.borderIt (@ad:0, @size:1px, @type: solid, @col: #000) {
    border: @size @type @col;
}

.borderIt (@ad:0) when (@ad > 0) {
    -webkit-border-radius: @ad;
    -moz-border-radius: @ad;
    border-radius: @ad;
}

#profiles {
    .borderIt(20px, 2px);
    .pad();
    th { .pad(); }
    td { .pad();
        .borderIt(0, 1px, solid, lighten(#000, 80%));
    }
}
```

We save this to `views/styles.less`. Express compiles it to `public/styles.css` and once again our `#profiles` table has rounded corners.

See also

- ▶ *Templating in Express* discussed in this chapter
- ▶ *Generating Express scaffolding* discussed in this chapter

Initializing and using a session

If we want to maintain state between page requests, we use sessions. Express supplies middleware that takes much of the complexity out of managing sessions. In this recipe, we're going to use Express to make a session between a browser and server to facilitate a user login process.

Getting ready

Let's create a fresh project:

```
express login
```

This will create a new Express skeleton named `login`.

How to do it...

In our `app.js` file, we make the following changes to `app.configure`:

```
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(express.cookieParser());
app.use(express.session({secret: 'kooBkooCedoN'}));
app.use(app.router);
app.use(express.static(__dirname + '/public'));
});
```

Sessions are dependent on cookies, so we need both `cookieParser` and `session` middleware.

Express 2 to Express 3

In Express 3, we set the secret string through `cookieParser` instead of `session`:

```
app.use(express.cookieParser('kooBkooCedoN'));
app.use(express.session());
```

We'll finish `app.js` with some route handling as follows:

```
app.get('/', routes.index);
app.post('/', routes.login, routes.index);
app.del('/', routes.logout, routes.index);
```

The GET requests will serve pages as normal, the POST requests will be interpreted as login attempts. These will first be passed to a validating route which checks for valid user data. The DELETE request will clear the session with `routes.logout` and then pass over to `routes.index`.

Now to edit the `routes/index.js` file:

```
var users = {'dave' : 'expressrocks'}; //fake user db:  
exports.login = function (req, res, next) {  
    var user = req.body.user;  
    if (user) {  
        Object.keys(users).forEach(function (name) {  
            if (user.name === name && user.pwd === users[name]) {  
                req.session.user = {  
                    name: user.name,  
                    pwd: user.pwd  
                };  
            }  
        });  
    }  
    next();  
};  
  
exports.logout = function (req, res, next) {  
    delete req.session.user;  
    next();  
}  
  
exports.index = function (req, res) {  
    res.render('index', {title: 'Express', user: req.session.user});  
};
```

Finally, let's put a login form together in a file. We'll call `login.jade` as follows:

```
if user  
form(method='post', action='/')  
    input(name="_method", type="hidden", value="DELETE")  
    p Hello #{user.name}!  
    a(href='javascript:', onClick='forms[0].submit()') [logout]  
  
else  
    p Please log in  
    form(method='post', action='/')  
        fieldset  
            legend Login
```

```
p
  label(for="user[name]") Username:
  input(name="user[name]")

p
  label(for="user[pwd]") Password:
  input(type="password", name="user[pwd]")

input(type="submit")
```

_method

Notice how our logout form uses a hidden input named `_method`. Setting this value to `DELETE` overrides the `POST` method that the form is set to. This is made possible by the `methodOverride` middleware inside `app.configure` within the `app.js` file.

We'll include this form within `index.jade`:

```
h1= title
p Welcome to #{title}

include login.jade
```

Now if we run our app, and navigate to `http://localhost:3000`, we'll see a login form. We enter username `dave`, password `expressrocks` and now we see a greeting with the option to logout.

How it works...

In order to use sessions we have to include some additional middleware. We do this within `app.configure.express.parseCookie` comes first because `express.session` is dependent on it.

`express.session` takes a mandatory object containing the `secret` property (or in Express 3 the secret is set by passing a string argument to `express.cookieParser`). `secret` is used to generate the session hash, so it needs to be unique and unknown to outsiders.

When we set up our routes, we assume that POST requests to the `/` path are login attempts, and thus pass them first to the `Login` route, and DELETE requests to `/` are to be processed primarily by the `Logout` route.

Our `Login` route checks the posted login details (using `req.body`, which is supplied to us by the `bodyParser` middleware) against our placeholder `users` object. In a real-world scenario, `Login` would rather be validating against a database of users.

If everything checks out, we add a `user` object to the session, and place the name and password (`pwd`) into it.

When pushing user details to the session, we could have taken a shortcut and said:

```
req.session.user = req.body.user;
```

However, doing so could leave us open for an attacker to fill the `req.session.user` object with anything they desire, in potentially large amounts. While any data being entered into session would be entered by a trusted user (one with login details), and although `bodyParser` has built-in safety limits for POST data, it is always better to err on the side of conservatism over convenience.

The `index` route remains the same, except that we set a `user` property, to which we pass `req.session.user`.

This enables `Login.jade` to check the `user` variable. If it is set, `Login.jade` shows a greeting along with a small form containing a link that sends a POST request with a `DELETE` override to the server, thus triggering the `Logout` route via `app.del`.

The `Logout` route simply deletes the `user` object from `req.session`, passes control to the `index.route` (using `next`), which pushes a non-existent `req.session.user` back to Jade via `res.render`.

When Jade finds that there is no `user` it displays the login form, which is of course also output to a pre-login request.

There's more...

We can improve the way we interact with sessions.

Custom middleware for site-wide session management

This recipe is fine if we want to pass our login and logout requests to just one route. However, as our routes and views increase, managing the complexities with sessions could become burdensome. We can mitigate this somewhat by creating our own custom middleware for session-handling purposes.

In preparation to test from different URLs, we'll modify our routes as follows:

```
app.get('/', routes.index);
app.post('/', routes.index);
app.del('/', routes.index);
app.get('/:page', routes.index);
```

We're not using routes to control our session logic, so we've removed the middle routes, sending them straight to `routes.index`. `:page` would probably point to another route, but we'll leave it as `routes.index` for brevity.

In `routes/index.js`, we can now simply have the following code:

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

Now let's create a file and call it `login.js`, writing the following code:

```
var users = {'dave' : 'expressrocks'};

module.exports = function (req, res, next) {
  var method = req.method.toLowerCase(), //cache the method
      user = req.body.user,
      logout = (method === 'delete'),
      login = (method === 'post' && user),

      routes = req.app.routes.routes[method];

  if (!routes) { next(); return; }

  if (login || logout) {
    routes.forEach(function (route) {
      if (!(req.url.match(route.regex))) {
        console.log(req.url);
        req.method = 'GET';
      }
    });
  }
  if (logout) {
    delete req.session.user;
  }

  if (login) {
    Object.keys(users).forEach(function (name) {
      if (user.name === name && user.pwd === users[name]) {
        req.session.user = {
          name: user.name,
          pwd: user.pwd
        };
      }
    });
  }
  if (!req.session.user) { req.url = '/'; }
  next();
};
```

Since we're no longer using routes, we don't have the opportunity to pass `req.session.user` through `res.render`. However, we can use a dynamic helper instead. Dynamic helpers have access to the `req` and `res` objects, they're called just before a view is rendered. Any properties we pass to the dynamic helper object are pushed to the Jade view as local variables. In `app.js`, just above our routes we put:

```
app.dynamicHelpers({
  user: function (req, res) {
    return req.session.user;
  }
});
```

Express 2 to Express 3

In Express 3, dynamic helpers are set with `app.locals.use`:



```
app.locals.use(function (req, res) {
  res.locals.user = req.session.user;
});
```

Rather than sending an object containing desired locals, local variables are explicitly set by adding them to the `res.locals` object.

Now we simply include `login.js` as middleware inside the `app.configure` callback of `app.js`:

```
app.configure(function()
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(express.cookieParser());
  app.use(express.session({secret: 'kooBkooCedoN'}));
  app.use(require('./login'));
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});
```

Finally, we'll modify `login.jade` so instead of:

```
form(method='post', action='/')
```

We have:

```
form(method='post')
```

This makes the form POST to whatever address it is submitted from.

Now all the muscle work is performed by `>Login.js`. The bottom half of the exported function performs the same actions as our recipe. We have to check the methods manually since we're not using the Express router.

In the top half we access `req.app`. In `app.js`, the `app` variable is the result of the `express.createServer`. Express allows us to access our server within middleware and routes with a reference to the server instance under `req.app`.

In our interaction with `req.app`, we work with `req.app.routes.routes` which we store in the `routes` variable. This property holds any routes we defined using `app.get`, `app.post`, and so on. The routes are stored by request method type, for example, `req.app.routes.routes.post` holds an array of all the `app.post` routes. If a route method hasn't been defined, we simply call `next` and `return`. This lets Express handle the problem of an undefined method. Each item in the array is an object, containing `path`, `method`, `callbacks`, `keys`, and `regexp` properties. We loop through all routes for the `request` method, and use the `regexp` property to determine if there is a `match` for the requested URL. If there isn't, we reset the method to `GET`. We do this to transparently ensure that the `POST` and `DELETE` requests can be served through any URL and not return a `404 error` if a `post` or `del` route hasn't been defined for them.

If this piece of code was missing, the login or logout mechanism would still occur, but the user would be left with a `not found` message. For instance, if we navigate to `http://localhost:3000/anypage`, and attempt to log in, our middleware will catch the request first. It will determine if login conditions are met (a `POST` request with `user` in the body) and processes it accordingly. If no `POST` route has been defined for `/anypage`, we reset the method to `GET`. The middleware later calls `next`, passing control over to `app.router`, which never sees the `POST` method and thus reloads the `/:page` `GET` route.

Back in `app.js` we have Express' `dynamicHelpers` method. The `dynamicHelpers` method registers the helper, but it isn't called until just before a view is rendered (which means the dynamic helper is executed after all our route callbacks). This is convenient because it allows our routes to interact with `req.session.user` further if required. We pass an object to `dynamicHelpers` containing a `user` property. The ultimate value of the `user` property is loaded directly into our view as a variable. In the same way we can pass variables to views via the `res.render` options object within a route. The `user` property holds a callback which is evaluated by Express. It works much like a route or middleware callback except a `return` value is expected. We `return req.session.user`, so as in the main recipe, `req.session.user` is now available in `Login.js` as `user`. If there is no session, we make sure to reset the URL to `/` so that other routes cannot be used to bypass our authorization process.

Finally, we call `next` which passes control to the next middleware, which in our case is `app.router`.

Flash messages

Express provides a simple interface for session-based flash messages. A flash message is held in a session object for one request, and then disappears. It's an easy way to generate request associated information or error messages.

Express 2 to Express 3

Express 3 doesn't support session flash messages out of the box. However, connect-flash provides this functionality as middleware.

In the dependencies section of our package.json file we would add:

```
, "connect-flash": "0.1.0"
```

And then do a:

```
npm install
```

Finally, we would require connect-flash and call it inside the app.

configure callback, after the cookie and session middleware:



```
var express = require('express')
, routes = require('./routes')
, flash = require('connect-flash')
var app = module.exports = express.createServer();

app.configure(function(){
// prior middleware
app.use(express.cookieParser('kooBkooCedoN'));
app.use(express.session());
app.use(flash());
// rest of app.configure callback
})
```

Let's modify our login.js file from the previous extension to our recipe (*Custom middleware for site wide session management*). We're going to modify it to flash an error message for invalid login details. First, we'll need to alter the code located at the bottom of our exported function, inside the if (login) conditional:

```
if (login) {
var valid = Object.keys(users).some(function (name) {
return (user.name === name && user.pwd === users[name]);
});
if (valid) {
req.session.user = {
name: req.body.user.name,
pwd: req.body.user.pwd
};
} else {
req.flash('error', 'Login details invalid!');
}
}
```

This works fine, but we can do better. Let's tidy it up by extracting our validation code into a separate function:

```
function validate(user, cb) {
  var valid = Object.keys(users).some(function (name) {
    return (user.name === name && user.pwd === users[name]);
  });
  cb(!valid && {msg: 'Login details invalid!' });
}


```

Although everything happening in `validate` is synchronous, we've written it in asynchronous style (that is, passing values through a callback instead of returning values). This is because, in reality, we wouldn't use an object to store user details. We would use a remote database, which would have to be accessed asynchronously. In the next recipe, we'll be storing our user details in a MongoDB database and asynchronously reading it to validate login requests. The `validate` function is structured with this in mind.

Now we replace our middleware user validation code with the following:

```
if (login) {
  validate(user, function (err) {
    if (err) { req.flash('error', err.msg); return; }
    req.session.user = {
      name: user.name,
      pwd: user.pwd
    };
  });
}
if (!req.session.user) { req.url = '/'; }
next();
}; // closing bracket of module.exports
```

Our `validate` function uses the same Boolean method. However, it's tucked out of the way. Also notice the various strategic calls to `next`—either when exiting early from an error, adding a user session, or at the very end. Placing these `next` calls within the callback context future proofs our validation function for asynchronous operations, which is important for database interactions.

We use the `callback(err)` style from the `validate` function to let our middleware know whether the login was successful. `err` is simply an object containing the error message (`msg`), it's only passed if `valid` is not true.

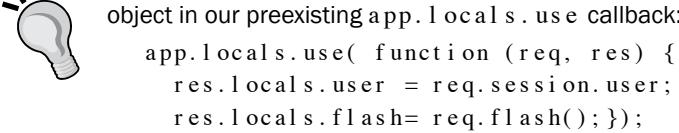
If `err` is present, we call `req.flash`, which is the built-in Express method that pushes an object called `flash` onto `req.session`. Once the request is fulfilled, the object is emptied of all properties.

We need to make this object available to `login.jade`, so we'll add another dynamic helper over in `app.js`.

```
app.dynamicHelpers({
  user: function (req, res) {
    return req.session.user;
  },
  flash: function (req, res) {
    return req.flash();
  }
});
```

Express 2 to Express 3

To add flash messages in Express 3, we'd simply add it to the `res.locals` object in our preexisting `app.locals.use` callback:



```
app.locals.use(function (req, res) {
  res.locals.user = req.session.user;
  res.locals.flash = req.flash();});
```

We could use `req.session.flash`, but `req.flash()` will do the same.

Finally, at the top of `login.jade` we write:

```
if flash.error
  hr
  b= flash.error
  hr
```

If login details are incorrect, the user receives a bold error notification between horizontal lines.

See also

- ▶ *Making an Express web app* discussed in this chapter
- ▶ *Templating in Express* discussed in this chapter
- ▶ *Dynamic routing* discussed in this chapter

Making an Express web app

In this recipe, we're going to combine a lot of previous recipes together, also throwing in a few extra Express features (such as app mounting) in order to create the foundations of an Express-based web app with integrated administration features.

Getting ready

Let's start fresh, from the command line we say:

```
express profiler
```

Profiler is the name of our new app, it will be a profile manager for members of the Node community.

We need to edit `package.json` to say:

```
{
  "name": "Profiler",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "2.5.8",
    "jade": ">= 0.0.1",
    "stylus": "0.27.x",
    "mongoskin": "0.3.6"
  }
}
```

We've set the name to `Profiler` adding `stylus` and `mongoskin`, setting stricter version requirements for `mongoskin`. Jade and Stylus are built to work with Express, so they're likely to remain compatible with new releases (although we've restricted Stylus to minor version updates). Mongoskin has its own development processes. To ensure our project isn't broken by future versions with possible API changes we've locked the version 0.3.6 (though this doesn't mean we can't upgrade at a later date).

So we get our dependencies with:

```
npm install
```

We also need to ensure MongoDB is installed and running on our system, see the *Storing and retrieving data with MongoDB* recipe of *Chapter 4, Interfacing with Databases*, for details.

In short, we start Mongo with the following command:

```
sudo mongod --dbpath [a folder for the database]
```

We'll also push some data into MongoDB to get us started. Let's create a new folder in the `profiler` directory, and call it `tools`. Then pull our `profiles.js` module from *Chapter 1, Making a Web Server*, into it, creating a new file called `prepopulate.js`. Inside it we write the following code:

```
var mongo = require('mongoskin'),
  db = mongo.db('localhost:27017/profiler'),
```

```

profiles = require('./profiles'),
users = [{name : 'dave', pwd: 'expressrocks'},
          {name : 'M Page', pwd: 'hellomynameismpage'}
        ];
// make sure collection is empty before populating
db.collection('users').remove({});
db.collection('profiles').remove({});

db.collection('users').insert(users);
Object.keys(profiles).forEach(function (key) {
  db.collection('profiles').insert(profiles[key]);
});

db.close();

```

When executed, this gives us a database named `profiler` with a `profiles` and `users` collection.

Finally, we'll be using the entire login app of the previous recipe. However, we'll want it with the site-wide session management and flash messages (in the code examples, this folder is called `login_flash_messages`). So let's copy the `login` folder to our new profile directory as `profiler/login`.

How to do it...

Creating a database bridge

Let's begin with some backend coding. We'll create a new folder called `models`, and create a file inside it called `profiles.js`. This is going to be used to manage all our interactions with the MongoDB `profiles` collection. In `models/profiles.js` we put:

```

var mongo = require('mongoskin'),
  db = mongo.db('localhost:27017/profiler'),
  profs = db.collection('profiles');

exports.pull = function pull(page, cb) {
  var p = {},
    //rowsPer = 10, //realistic rowsPer
    rowsPer = 2,
    skip, errs;
  page = page || 1;
  skip = (page - 1) * rowsPer;

  profs.findEach({}, {limit: rowsPer, skip: skip},
    function (err, doc) {
      if (err) { errs = errs || []; errs.push(err); }
    }
  );
}

```

```
        if (doc) {
          p[ doc._id] = doc;
          delete p[ doc._id]. _id;
          return;
        }
        cb(errs, p);
      });
    }

exports.del = function (profile, cb) {
  profs.removeById(profile, cb);
}

exports.add = function (profile, cb) {
  profs.insert(profile.profile, cb);
}
```

We've defined three methods: `pull`, `del`, and `add`. Each one operates on the database asynchronously, and executes a user callback once data is returned, or operation is complete. We've set a low rows-per-page limit (`rowsPer`) to allow us to test our pagination work (the dividing of content into pages) with the few records we have.

We must also modify `login/login.js`, which we created in the previous recipe, to hook up our login app to the MongoDB user collection. The main module can remain untouched. We only have to change the way we validate a user, everything above `module.exports` changes to:

```
var mongo = require('mongoskin'),
  db = mongo.db('localhost:27017/profiler'),
  users = db.collection('users');

function validate(user, cb) {
  users.findOne({name: user.name, pwd: user.pwd},
    function (err, user) {
      if (err) { throw err; }
      if (!user) {
        cb({msg: 'Invalid login details!'});
        return;
      }
      cb();
    });
}

}
```

Configuring app.js files

Now let's modify `app.js`.

`app.configure` should look like this:

```
app.configure(function() {
    app.set('views', __dirname + '/views');
    app.set('view engine', 'jade');

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

    app.use(require('stylus').middleware({
        src: __dirname + '/views',
        dest: __dirname + '/public'
    }));

    app.use(express.favicon());

    app.use(app.router);
    app.use(express.static(__dirname + '/public'));

    app.use('/admin', require('./login/app'));
});
```

We've fired up the Stylus engine and thrown in a favicon server for good measure. The last line `app.use` actually mounts our login app at the `/admin` route (we copied `login` into our profiler directory in *Getting ready*).

Next, let's add our one and only route to our main `app.js` as follows:

```
app.get('/:pagenum([0-9]+)?', routes.index);
```

We've specified an optional placeholder called `:pagenum`, which must consist of one or more digits. So `/`, `/1`, `/12`, and `/125452` would all be valid routes, but `/alphaCharsPage` would not.

Now we'll set up some extra configuration details in the `app.js` of our `login` app in `login/app.js` as shown in the following code:

```
app.configure(function() {
    app.set('views', __dirname + '/views');
    app.set('view engine', 'jade');
    app.use(express.bodyParser());
    app.use(express.methodOverride());
    app.use(express.cookieParser());
    app.use(express.session({secret: 'kooBkooCedoN'}));
```

```
app.use(require('stylus').middleware({
  src: __dirname + '/views',
  dest: __dirname + '/public'
}));

app.mounted(function (parent) {
  this.helpers({masterviews: parent._locals.settings.views + '/'});
});

app.use(require('./login'));
app.use(app.router);
app.use(express.static(__dirname + '/public'));
});
```



Express 2 to Express 3

Remember in Express 3, the secret goes in the `express.cookieParser` as a string instead of inside an object passed into `express.session`.

The `login` app will be pulling in our `profiles` table from the `profiler` app. We've configured it to use Stylus as we'll be applying extra admin-specific Stylus-generated CSS. We've also added a helper variable called `masterviews`. This will later be used to locate the absolute path of our app's main views directory. The `login` app will need to know this in order to load views from its parent `profiler` app.

Next, we'll modify the routes in our `login/app.js` file:

```
app.get('/:pagenum([0-9]+)?', routes.index);
app.post('/:pagenum([0-9]+)?', routes.index);
app.del('/:pagenum([0-9]+)?', routes.index);

app.get('/del', routes.delprof);
app.post('/add', routes.addprof, routes.index);
```

Adding the optional `:pagenum` to the `get` method route enables navigation of the `profiles` table as in the main app. Adding `:pagenum` to the `post` method route allows users to log in from pages they may have previously navigated to, (for example, this allows a login form to be served from `http://localhost/admin/2` if a user's session is expired). Likewise, the `del` method route will allow us to log out from any valid page.

We've also added a `/del` and `/add` route for processing admin tasks.

When one Express app is mounted inside another, calling the `listen` method on the sub-app causes a port conflict. Sub-apps do not have to listen, their parent app does the listening for them.

So we modify the `listen` invocation of `login/app.js` to:

```
if (!module.parent) {
  app.listen(3000, function() {
    console.log("Express server listening on port %d in %s mode",
    app.address().port, app.settings.env);
  });
}
```

`module` is a built-in Node global. The `parent` property tells us whether or not our app is being loaded by another app. Since our login app is being loaded by the profiler app, `app.listen` won't trigger.

The login app is the gatekeeper of our administration section within which we will be able to add and remove profiles.

Now that our main app and mounted app are suitably prepared, we can move onto editing our views, styles, and routes.

Modifying the profiler app

Let's start with the `profiler/app/index.jade` view.

```
h1= title
p Welcome to #{title}
p: a(href='/admin/') [ Admin Login ]
include profiles
```

Since we're including `profiles.jade`, let's write it as follows:

```
masterviews = typeof(masterviews) !== 'undefined' ? masterviews : ''
table#profiles
tfoot
  page = (page) || 1
  s = (page > 1) ? null : 'display:none'
  td
    a#bck(href="{{(+page-1)}}", style=s) &quo;
    a#fwd(href="{{(+page+1)}}") &quo;
thead
  tr
    th Name
    th Irc
    th Twitter
    th Github
    th Location
    th Description
  if typeof user !== 'undefined'
```

```
    th Action
  t body
    each profile, id in profiles
      tr(id=id)
        != partial(masterviews + 'row', {collection: profile})
        mixin del(id)
  mixin add

mixin adminScript
```

Express 2 to Express 3

We're using a `partial` (see *Jade Partials* discussed in *Templating in Express*) in `profiles.jade`. Express 3 no longer supports partials so we would need to iterate through the rows manually instead:



```
  t body
    each profile, id in profiles
      tr(id=id)
        each row in profile
          td= row
        mixin del(id)
```

`profiles.jade` should be saved to the `profile/views` directory, it is based on our `profiles` table in the previous recipes. However, we've added code to support seamless integration with the login app and some additional HTML structure for pagination.

At the top of `profiles.jade`, we've included a safety net to ensure our view doesn't choke if `masterviews` is undefined. For pagination, we've added a `tfoot` element to hold the back and forward links with a complimentary `thead` to hold wrap the `th` elements.

We're using a partial to load each row, this will be loaded from `row.jade` as follows:

```
td= row
```

We've also included some Jade mixin invocations. We'll be defining these mixins when we come to edit the login apps views.

Let's create a new `stylesheets` directory under `views` and place a file in it called `style.css`.

In `views/stylesheets/style.css` we write the following code:

```
body
  padding 50px
  font 14px "Lucida Grande", Helvetica, Arial, sans-serif;
a
  color #00B7FF
```

```
rndCorner(rad, sides...)
if length(sides) is 2
  -moz-border-radius-{sides[0]}{sides[1]} rad
  -webkit-border-{sides[0]}-{sides[1]}-radius rad
  border-{sides[0]}-{sides[1]}-radius rad
else
  -webkit-border-radius rad
  -moz-border-radius rad
  border-radius rad

borderIt(rad = 0, size = 1px, type = solid, col = #000)
border size type col
if rad {rndCorner(rad)}

.pad
padding 0.5em

tdRad = 9px

#profiles
width 950px
borderIt 20px, 2px
@extend .pad
background: #000;
color: #fff;
th
@extend .pad
tbody
td
@extend .pad
background blue + 35%
borderIt(col: @background)
color pink - green - brown + gold - green
color desaturate(@color + 100, 100)
& hover
color @background + 180deg
background desaturate(@background, 40)
border-color @background

tr
& first-child
td:first-child
rndCorner tdRad top left
```

```
td:last-child
  roundCorner tdRad top right
&last-child
  td:first-child
  roundCorner tdRad bottom left
  td:last-child
  roundCorner tdRad bottom right

tfoot
  font-size 1.5em
  td
    a
      textDecoration none
      color #fff - 10%
      & hover
        color #fff
```

This is the same Stylus sheet from the *Playing with color* section under *There's more...* in the *CSS engines with Express* recipe, but with some modifications.

Since we've placed our `th` elements under a `thead`, we can simply select our `tbody tr` elements by `:first-child` instead of `:nth-child(2)`. We also add some styling for the new `tfoot` element.

Finally, we'll write the `routes/index.js` file.

```
var profiles = require('../models/profiles');

function patchMixins(req, mixins) {
  if (!req.session || !req.session.user) {
    var noop = function() {},
        dummies = {};
    mixins.forEach(function (mixin) {
      dummies[mixin + '_mixin'] = noop;
    });
    req.app.helpers(dummies);
  }
}

exports.index = function (req, res) {
  profiles.pull(req.params.pageNum, function (err, profiles) {
    if (err) { console.log(err); }
    // output no-ops to avoid choking on non-existent admin mixins
    patchMixins(req, ['add', 'del', 'adminScript']);
  });
}
```

```

    res.render('index', { title: 'Profiler', profiles: profiles,
        page: req.params.pageNum });
    });
};

```

Our `index` route makes a call via our `models/profiles.js` module to MongoDB, passing it the desired page number, and retrieves some profiles to display.

It also calls our `patchMixins` function, included prior to our route, with an array of mixin names found in `profiles.jade`. These mixins don't exist yet. Furthermore, the mixins will only be available when we're logged in to `http://localhost:8080/admin`. This is intentional, the mixins will provide administrative controls which lie on top of our `profiles` table, we only want them to appear when a user is logged in.

However, if we don't include dummy mixins in the place of the admin mixins, Node will throw an error.

Internally, Jade mixins are compiled into JavaScript functions before they are executed within our view templates. So we create dummy no-op (no operation) functions to prevent a server error. Then when we do log in, they are replaced with the administration mixins.

If we navigate to `localhost:3000` we should now have a functioning `profiler` app.

Modifying the mounted login App

In `login/views`, we currently have `index.jade`, `login.jade`, and `layout.jade`. In `login.jade`, we want to add two includes as follows:

```

if flash.error
    hr
    b= flash.error
    hr

if user
    form(method='post')
        input(name='_method', type="hidden", value="DELETE")
        p Hello #{user.name}!
        a(href='javascript:', onClick='forms[0].submit()') [logout]

    include admin
    include ../../views/profiles

else
    p Please log in

// rest of the login.jade...

```

Rather than repeating code, we reuse our `profiles.jade` view from the main app using a relative path. This means any changes we make to our frontend site are also made to our administration section! `admin.jade` is going to contain Jade mixins (which are conceptually similar to Stylus mixins). These mixins are conditionally included in `profiles.jade` (see the previous *Modifying the profiler app* section).

So in `admin.jade`:

```
mixin del(id)
  td
    a.del(href='/admin/del?id=#{id}&p=#{page}')
      &#10754;

mixin add
  #ctrl
    a#add(href='#') &oplus;

mixin adminScript
  include adminScript

include addfrm
```

We have two includes in `admin.jade`, one as part of a mixin, and the other as a straight include.

`addfrm.jade` should be as follows:

```
fields = ['Name', 'Irc', 'Twitter', 'Github', 'Location',
'Description'];

form#addfrm(method='post', action='/admin/add')
  fieldset
    legend Add
    each field, i in fields
      div
        label=field
        input(name="profile[#{field.toLowerCase()}]")
      .btns
        button.cancel(type='button') Cancel
        input(type='submit', value='Add')
```

And `adminScript.jade` should contain the following code:

```
script(src='https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.
min.js')
script
  document.getElementsByTagName('body')[0].id = 'js';
```

```
$(' #add' ).click(function (e) {
  e.preventDefault();
  $(' #profiles, #ctrl' ).fadeOut(function () {
    $(' #addfrm' ).fadeIn();
  });

  $(' #addfrm .cancel' ).click(function () {
    $(' #addfrm' ).fadeOut(function () {
      $(' #profiles, #ctrl' ).fadeIn();
    });
  });
});
```

Admin is positioned above profiles in `login.jade`, so `#addfrm` will sit above the `#profiles` table. However, our `adminScript` mixin hides the table, showing it when the **Add** button is clicked.

We create `stylesheets` folder under `login/views`, creating `admin.style.styl` in it and write the following code:

```
@import '/views/stylesheets/style.styl'

tbody
  td
    .del
      text-decoration none
      color blue + 35% + 180deg
      float right
      & hover
        color red
  #ctrl
    width 950px
    text-align center
    margin-top -2.5em
    a
      color white - 10%
      font-size 1.8em
      text-decoration none
      & hover
        color @color + 111%

#js
  #addfrm
    display none
```

```
#addfrm
width 250px
label
  display block
  float left
  width 100px
  font-weight bold
.buttons
  width @width
  text-align right
```

Now we're also reusing the Stylus sheet from our main app. The `@import` declarative is handled by Stylus on the server side (unless the extension is `.css`). As a result, our main app's `styles.styl` sheet is combined with `admin.styl` and compiled as one CSS file in `login/public/stylesheets/admin.css`.

To load our `admin.css` file, we must alter the login app's `layout.jade` view as shown in the following code:

```
!!!
html
  head
    title= title
    link(rel='stylesheet', href='/admin/stylesheets/admin.css')
  body!= body
```

We've altered the `link href` attribute from `/stylesheets/style.css` to `/admin/stylesheets/admin.css`, ensuring that CSS is loaded from the static server on our sub-apps route.

Finally, we complete our admin routes, in `login/routes/index.js` as follows:

```
var profiles = require('../models/profiles');

exports.index = function (req, res) {
  profiles.pull(req.params.pagenum, function (err, profiles) {
    if (err) { console.log(err); }
    res.render('index', { title: 'Profiler Admin', profiles: profiles,
      page: req.params.pagenum });
  });
};

exports.delprof = function (req, res) {
  profiles.del(req.query.id, function (err) {
    if (err) { console.log(err); }
```

```

    profiles.pull(req.query.p, function (err, profiles) {
      req.app.helpers({profiles: profiles});
      res.redirect(req.header('Referrer') || '/');
    });
  });

  exports.addprof = function (req, res) {
    profiles.add(req.body, function (err) {
      if (err) { console.log(err); }
      res.redirect(req.header('Referrer') || '/');
    });
  };
}

```

We should now be able to log in to `http://localhost:3000/admin` to delete and add profiles as `Dave`, with password `expressrocks`, or as `M. Page`, with password `hello my names mr page`.



Login security

In the next chapter, we will learn how to hash our passwords and log in over SSL.

How it works...

Our app contains a lot of pieces working together. So let's look at it from various angles.

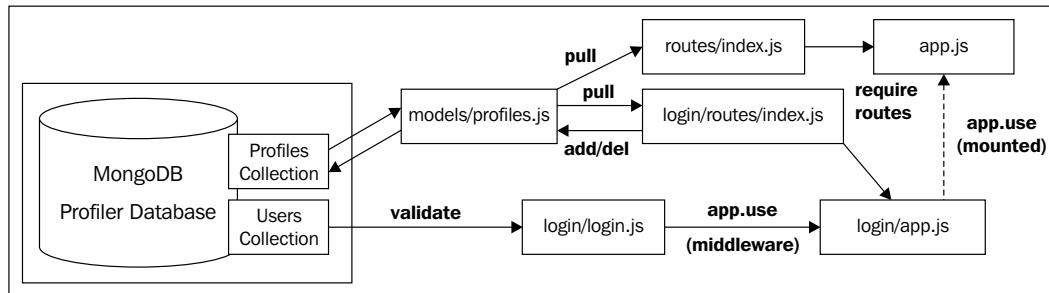
App mounting

In this recipe, we have two apps working with the same database, sharing views and Stylus sheets. We imported the login app to our new `profile` folder and mounted it with `app.use` setting `/admin` as its route.

This works because Express apps are an assemblage of middleware, so when we mount the login app it simply integrates with our app as a middleware plugin. Middleware works on the request and response objects. By passing the `/admin` route into `app.use`, we limit the login app to work only with requests made under that route.

Data Flow

Our app is supported by a MongoDB database that we set up with our `prepopulate.js` tool. Data flows to and from the database as shown:



`models/pr o f i l e s . j s` pulls and pushes data to the profiles collection, providing an interface for the `r ou t e s / i n d e x . j s` files in both master and sub-apps. Our routes integrate within their respective `app. j s` files, and work to interact with `models/pr o f i l e s . j s` to carry out the desired tasks.

`l o g i n . j s` simply verifies the user's credentials, performing a search with user-supplied input. `l o g i n . j s` sits as a piece of middleware within `l o g i n / a p p . j s`, waiting to respond to POST requests containing a username and password.

Route handling

In both apps, the `i ndex` route provides a foundation for displaying and navigating the profiles table. In both we call `pr o f i l e s . p u l l`, passing `i n r e q . p a r a m s . p a g e n u m`. The `p a g e n u m` parameter is loaded onto `r e q . p a r a m s`. It will never be anything but a number—thanks to our restrictions placed on it, though it is optional so may not be present.

Our `pr o f i l e s . p u l l` takes two parameters: the page number and a callback. If the page number isn't present, it sets the page to 1. We determine which rows to extract by multiplying our internal `r o w s P e r` variable by `page - 1` (we want to start at the beginning with page 1, therefore, for the first page we skip 0 rows). The result is passed through as the `s k i p` modifier to MongoDB, as well as `r o w s P e r` being passed as the `l i m i t` property. `s k i p` will pass over a predetermined number of rows before outputting and `l i m i t` restricts the amount to output; thus we achieve pagination.

The `pr o f i l e s . p u l l` callback is initiated either with an error, or with an object containing profiles. In both our `i ndex` routes, we perform minimal error handling. Express tends to capture the errors and output them to the browser for debugging purposes. `pr o f i l e s` is passed to `r e s . r e n d e r` where it is later utilized by the `pr o f i l e s . j a d e` view.

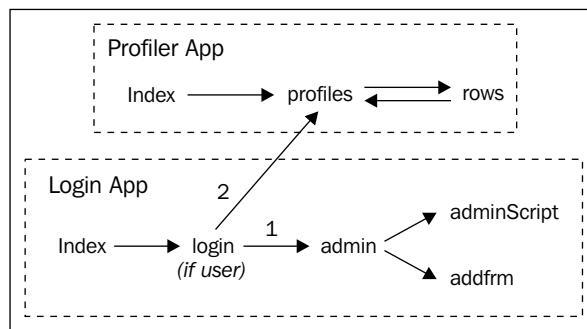
In `login/app.js`, two inflexible routes are defined: `/add` and `/del`. The `/del` route is a basic GET request pointing to `routes.delprof`, it expects two query parameters: `i d` and `p.` `i d` is passed to `profiles.del`, which calls the Mongoskin `removeById` method, effectively deleting a profile from the collection. We pass the `c b` parameter directly to the `removeById` callback, making the `profiles.del` callback a direct result of `removeById`.

Back in `login/routes/index.js`, as long as no error has occurred, we invoke `profiles.pull`, with the `p` query parameter, updating the `profiles` object to the views using `app.helpers`. This ensures that changes to the database are reflected to the user. Finally, we redirect the user back to where they came from.

The `/add` route works in much the same way, except as a POST request. The `req` body is returned as an object, we can simply insert this object straight into MongoDB (since it is JSON-like).

Views

We use a lot of `includes` in our views, and sometimes between apps, the relationships look as shown in the following diagram:



In our main app, the index view loads the profiles view, and profiles utilizes the rows view in a `partial` statement.

In the login app, the index view includes the login view. The login view loads the profiles view and, under the right conditions, also includes the admin view (before profiles) to provide the administration layer. Admin includes the `addfrm` and `adminScript` views. The mixins defined in admin become available to profiles.

The `profiles.jade` view is very central to the entire web app, it outputs our data, delivers the optional administration overlay, and provides navigational capabilities. Let's take a look at the navigational portion:

```

table#profiles
  tfoot
    page = (page) || 1

```

```
s = (page > 1) ? 'display: none'  
t d  
a#bck(href="#{settings.basepath}/#{{( +page - 1)}}", style=s) &quo;  
a#fwd(href="#{settings.basepath}/#{{( +page + 1)}}") &quo;
```

The `page` variable is passed through from the `index` route, and is determined from `req.params.pageNum`. If `page` is 0 (or false), we set it to 1, which in the user's mind is the first page. We then create a variable named `s`. In Jade, we don't have to use `var`, Jade takes care of the complications. If we are on the first page, a link to the previous pages is unnecessary, so add a `style` attribute containing `display: none` (if we wanted to be neater, we could have a CSS class set `display` and add a `class` attribute instead). By passing `null`, if the `page` is greater than one, we're telling Jade that we don't want to set the `style` attribute at all.

Mixins

The only place we use Jade mixins is in the `login/views/admin.jade` view, but they are essential to the synergy between the admin section and the top-level site. Unless a user is logged in and under the `/admin` route, the mixins are not present in `profiles.jade`. They are only intended for privileged users.

We use mixins to supplement the `profiles` table with an administration layer. The only part of `admin.jade` that isn't a mixin is the final `include addfrm.jade`. As `admin` is included before `profiles`, `#addfrms` sits above the `profiles` table.

The `adminScript` mixin is, as the name suggests, a `script` block that quickly applies an `id` of `js` to the body tag. We use this in `admin.style` to hide our `#addfrm` (the generated CSS would be `#js #addfrm {display: none}`). This is quicker than directly hiding the element with JavaScript and minimizes the undesirable flash of content effect that can occur when hiding page elements on page load. Therefore, `#addfrm` is not initially visible. In the following screenshot, we can see the visible mixins displayed on the `#profiles` table within the admin section:

The screenshot shows a web application titled "Profiler Admin". At the top, there is a welcome message "Welcome to Profiler Admin" and a "Logout" link. Below this is a table with the following data:

Name	Irc	Twitter	Github	Location	Description	Action
Ryan Dahl	ryah	ryah	ry	San Francisco, USA	Creator of node.js	
TJ Holowaychuk	tjholowaychuk	tjholowaychuk	visionmedia	Victoria, BC, Canada	Author of express, jade and other popular modules	

At the bottom of the table, there is a black footer bar with a plus sign icon labeled "mixin add" and a minus sign icon labeled "mixin del".

Clicking on the **Add** button causes the `#profile` table to fade out and `#addform` to fade in. The `de1` mixin takes an `id` argument, which it then uses to generate a link for each profile, such as `/de1?id=4f3336f369cca0310e000003&p=1`. The `p` variable is determined from the `page` property passed in `atres.render` time in the `index` route.

Helpers

We use both static and dynamic helpers in the login app. Dynamic helpers sit between the final piece of route middleware and rendering of the view. So they have the last call on what is sent out. The dynamic helpers in our app stay the same.

Static helpers are set when the app spins up and can be overwritten at any time. These helpers are stored within `app.locals`, along with other Express presets (such as the `settings` object we use for our `base` variable in the `profiles` view). We use `app.mounted` within our login app to access the parent app object to discover our parent app's view directory from `parent._locals.settings.views`. Then we pass that back as the `masterviews` helper to our login app's views.

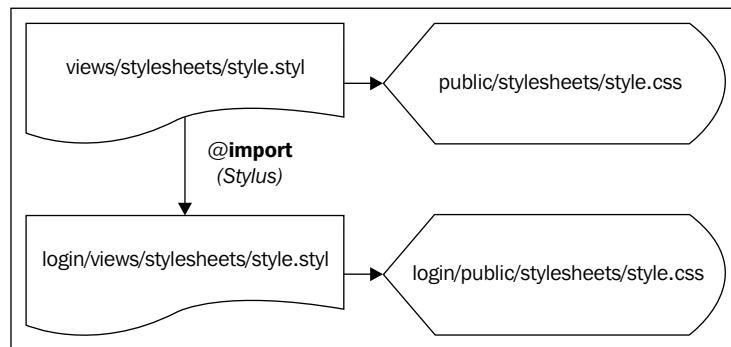
In `profiles.jade` we do this:

```
masterviews = typeof(masterviews) !== 'undefined' ? masterviews : ''
```

If, in our sub-app, we include a view from the parent app's and that view includes another parent view, or loads a partial with a parent view, we can use `masterviews` to ensure that the partial is loaded from the parent directory. `masterviews` enables `profiles.jade` to operate in both realms.

Styles

Our Stylus files also share a degree of interconnectivity, as shown in the following diagram:



User flow

All of these things work together to provide a website with an admin section.

A user can browse the profiles table, using the back and forward links, and they can link to a particular page on the table.

A privileged user can navigate to / admin, enter their login details, and proceed to add and delete records. The / add and / delete routes are protected by the middleware. Unless the user is logged in, the only route that gets delivered to them is the login app's index route asking for login details.

There's more...

Let's look at ways to monitor and analyze our web app.

Benchmarking

Benchmarking a Node website can be very satisfying, but there's always going to be room for improvement.

Apache Bench (ab) comes bundled with Apache servers, and while Apache have no part in NodeJS, their HTTP benchmarking utility is an excellent tool for stress testing our app's ability to respond to a large amount of simultaneous requests.

We can use it to test the performance benefits or hindrances of any changes to our app. Let's quickly throw 1,000 requests, 50 at a time, at both the site and the admin section as follows:

```
ab -n 1000 -c 50 http://localhost:3000/  
ab -n 1000 -c 50 http://localhost:3000/admin
```

Mileage will vary depending on system capabilities. However, since tests are run on the same machine, conclusions can be drawn from differences between tests.

With our test on the two sections, / delivers at 120 requests per second, whereas / admin serves at just under 160 request per second. This makes sense because the / admin page will only be serving a login form, whereas the / route is pulling data from MongoDB, performing iteration logic on the profiles object, and using a partial to display the rows.

Using logger

Express comes with Connect's logger middleware, which outputs useful custom-defined information from our app. In a production scenario, this information can be an essential part of site maintenance.

To use logger, we include it before all other middleware as follows:

```
app.configure(function(){  
  app.set('views', __dirname + '/views');
```

```

app.set('view engine', 'jade');

app.use(express.logger());

app.use(express.bodyParser());
app.use(express.methodOverride());
// rest of configuration

```

By default, logger outputs something as shown in the following code:

```

127.0.0.1 - - [Thu, 09 Feb 2012 15:56:40 GMT] "GET / HTTP/1.1" 200 908
"- Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.3 Safari/535.19"
127.0.0.1 - - [Thu, 09 Feb 2012 15:56:40 GMT] "GET /stylesheets/
style.css HTTP/1.1" 200 1395 "http://localhost:3000/" "Mozilla/5.0
(X11; Linux x86_64) AppleWebKit/535.19 (KHTML, like Gecko)
Chrome/18.0.1025.3 Safari/535.19"

```

It contains useful information about the request, the IP address of the client, the user agent string, and the content length (908 and 1395 respectively in the example output). We can use this information to determine a host of facts (for example, browser stats, geo-locational stats, and so on).

During development, we may want to set logger as shown in the following code:

```
app.use(express.logger('dev'));
```

This outputs color-formatted request information to the console, containing the request method (GET, for instance), the requested route, status number (such as 200), and time for the request to complete.

We can also pass in tokens. For example:

```
app.use(express.logger(':date -- :url :referrer::referrer'));
```

We can even define our own tokens as shown in the following code:

```

express.logger.token('external-referrer', function (req, res) {
  var ref = req.header('Referrer'),
      host = req.headers.host;

  if (ref && ref.replace(/https?:\/\/\//, '').split('/')[0] !==
host) {
    return ref;
  } else {
    return '#local#';
  }
});
app.use(express.logger(':date -- :url :referrer::external-
referrer'));

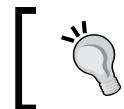
```

If the referrer is the same as our host, we mask it with `#local#`. Later, we could filter all lines containing `#local#`.

By default, `logger` outputs to the console. However, we can pass it a stream to output to. `Logger` would allow us to stream logs over TCP, HTTP, or simply to a file. Let's make our logs go to a file:

```
app.use(express.logger( //writing external sites to log file
  {format: ':date -- :URL: :referrer',
   stream: require('fs').createWriteStream('external-referrers.log')}));
});
```

This time we give `logger` an object instead of a string. In order to set the formatting through the object we set the `format` property. To redirect the stream of output, we create a `writableStream` to our desired log file.



For more logger options, see <http://www.senchalabs.org/connect/middleware-logger.html>.

See also

- ▶ *Dynamic routing* discussed in this chapter
- ▶ *Templating in Express* discussed in this chapter
- ▶ *CSS engines with Express* discussed in this chapter
- ▶ *Initializing and using a session* discussed in this chapter

7

Implementing Security, Encryption, and Authentication

In this chapter we will cover:

- ▶ Implementing Basic Authentication
- ▶ Cryptographic password hashing
- ▶ Implementing Digest Authentication
- ▶ Setting up an HTTPS web server
- ▶ Preventing cross-site request forgery

Introduction

When it comes to production web servers, security is paramount. The importance of security correlates with the importance of the data or services we provide. But even for the smallest projects, we want to ensure our systems aren't vulnerable to attack.

Many web development frameworks provide built-in security, which is a two-sided coin. On one side, we don't have to overly concern ourselves with the details (except for the basics, like cleaning user input before passing it into an SQL statement), but on the other we implicitly trust that the vendor has plugged all the holes.

If a largely used server-side scripting platform, such as PHP, is discovered to contain a security vulnerability, this can become public knowledge very quickly and every site running the vulnerable version of that framework is open to attack.

With Node, server-side security is almost entirely on our shoulders. Therefore, all we need to do is educate ourselves on the potential vulnerabilities and harden our systems and code.

For the most part, Node is minimalistic: if we don't specifically outline something it doesn't happen. This leaves little room for exploitation of unknown parts of our system or obscure configuration settings because we coded and configured our system by hand.

Attacks take place from two angles: exploiting technical flaws and taking advantage of user naiveté. We can protect our systems by educating ourselves and conscientiously checking and rechecking our code. We can also protect our users by educating them.

In this chapter, we will learn how to implement various types of user-authenticated logins, how to secure these logins, and encrypt any transferred data, along with a technique for preventing authenticated users from falling victim to exploits of the browser's security model.

Implementing Basic Authentication

The Basic Authentication standard has been in place since the 1990s and can be the simplest way to provide a user login. When used over HTTP, it is in no way secure since a plain text password is sent over the connection from browser to server.



For information on Basic Authentication see
http://en.wikipedia.org/wiki/Basic_authentication.



However, when coupled with SSL (HTTPS), Basic Authentication can be a useful method if we're not concerned about a custom-styled login form.



We discuss SSL/TLS (HTTPS) in the *Setting up an HTTPS web server* recipe of this chapter. For extra information see
<http://en.wikipedia.org/wiki/SSL/TLS>.



In this recipe, we'll learn how to initiate and process a Basic Access Authentication request over plain HTTP. In following recipes, we'll implement an HTTPS server, and see an advancement of Basic Authentication (Digest Authentication).

Getting ready

We just need to create a new `server.js` file in a new folder.

How to do it...

Basic Authentication specifies a username, password, and realm, and it works over HTTP. So we'll require the HTTP module, and set up some variables:

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

var username = 'dave',
password = 'ILikeBrie_33',
realm = "Node Cookbook";
```

Now we'll set up our HTTP server:

```
http.createServer(function (req, res) {
  var auth, login;

  if (!req.headers.authorization) {
    authenticate(res);
    return;
  }

  // extract base64 encoded username:password string from client
  auth = req.headers.authorization.replace(/^Basic /, '');
  // decode base64 to utf8
  auth = (new Buffer(auth, 'base64')).toString('utf8');

  login = auth.split(':'); // [0] is username [1] is password

  if (login[0] === username && login[1] === password) {
    res.end('Someone likes soft cheese!');
    return;
  }

  authenticate(res);
}) .listen(8080);
```

Notice we make two calls to a function named `authenticate`. We need to create this function, placing it above our `createServer` call:

```
function authenticate(res) {
  res.writeHead(401,
    {'WWW-Authenticate': 'Basic realm="' + realm + '"'});
  res.end('Authorization required.');
}
```

When we navigate to `localhost : 8080` in our browser we are asked to provide a username and password for the Node Cookbook realm. If we provide the correct details, our passion for soft cheese is revealed.

How it works...

Basic Authentication works via a series of headers sent between server and browser. When a browser hits the server, the `WWW-Authorization` header is sent to the browser and the browser responds by opening a dialog for the user to login.

The browser's login dialog blocks any further content from being loaded in the browser until the user either cancels or attempts to log in. If the user presses the **Cancel** button, they see the **Authorization required** message sent with `res.end` in the `authenticate` function.

However, if the user attempts to log in, the browser sends another request to the server. This time it contains an `Authorization` header in response to the `WWW-Authorization` header. We check for its existence at the top of the `createServer` callback with `req.headers.authorization`. If the header exists, we skip the call to `authenticate` and go on to verify the user credentials. The `Authorization` header looks like this:

```
Authorization: Basic ZGF2ZTpJTGlrZUJyaWfMzM=
```

The text following `Basic` is a Base64-encoded string that holds the username and password separated by a colon, so the decoded Base64 text is:

```
dave:I Like Br i e_33
```

In our `createServer` callback, we decode the Base64 header by first stripping the `Basic` portion from it, load it into a buffer which converts Base64 to binary, then run `toString` on the result converting it to a UTF8 string.

See <http://en.wikipedia.org/wiki/Base64> and http://en.wikipedia.org/wiki/Comparison_of_Uncode_encodings for information on Base64 and string encodings like UTF-8.

Finally, we split the login details with a colon, and if the provided username and password match our stored credentials, the user is granted access to authorized content.

There's more...

Basic Authentication comes bundled with the Express framework as middleware.

Basic Authentication with Express

Express (via Connect) provides the `basicAuth` middleware, which implements this pattern for us. To implement the same in Express:

```

var express = require('express');

var username = 'dave',
    password = 'I LikeBrie_33',
    realm = "Node Cookbook";

var app = express.createServer();

app.use(express.basicAuth(function (user, pass) {
    return username === user && password === pass;
}, realm));

app.get('/:route?', function (req, res) {
    res.end('Somebody likes soft cheese!');
});

app.listen(8080);

```

If we now head to `http://localhost:8080`, our Express server will behave in the same way as our main recipe.



See *Chapter 6, Accelerating Development with Express*, for information on using Express to develop web solutions.

See also

- ▶ *Setting up a router* discussed in *Chapter 1, Making a Web Server*
- ▶ *Implementing Digest Authentication* discussed in this chapter
- ▶ *Setting up an HTTPS web server* discussed in this chapter

Cryptographic password hashing

Effective encryption is a fundamental part of online security. Node provides the `crypt` module which we can use to generate our own MD5 or SHA1 hashes for user passwords. Cryptographic hashes, such as MD5 and SHA1 are known as message digests. Once the input data has been digested (encrypted), it cannot be put back into its original form (of course if we know the original password, we can regenerate the hash and compare it to our stored hash).

We can use hashes to encrypt a user's password before we store them. If our stored passwords were ever stolen by an attacker, they couldn't be used to log in because the attacker would not have the actual plain text passwords. However, since a hash algorithm always produces the same result, it could be possible for an attacker to crack a hash by matching it against hashes generated from a password dictionary (see the *There's more ...* section for ways to mitigate this).



See http://en.wikipedia.org/wiki/Cryptographic_hash_function for more information on hashes.



In this example, we will create a simple registration form, and use the `crypto` module to generate an MD5 hash of a password gained via user input.

As with Basic Authentication, our registration form should be posted over HTTPS, otherwise the password is sent as plain text.

Getting ready

In a new folder, let's create a new `server.js` file along with an HTML file for our registration form. We'll call it `register.html`.

We'll use the Express framework to provide the peripheral mechanisms (parsing POST requests, serving `register.html`, and so on), so Express should be installed. We covered more about Express and how to install it in the previous chapter.

How to do it...

First, let's put together our registration form (`register.html`):

```
<form method="post">
  User <input name="user">
  Pass <input type="password" name="pass">
  <input type="submit">
</form>
```

For `server.js`, we'll require `express` and `crypto`. Then create our server as follows:

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

var userStore = {},
app = express.createServer().listen(8080);

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

`bodyParser` gives us POST capabilities and our `userStore` object is for storing registered user details. In production we would use a database.

Now to set up a GET route as shown in the following code:

```
app.get('/', function (req, res) {
  res.sendFile('regform.html');
});
```

This uses Express' `sendfile` method to stream our `regform.html` file.

Finally, our POST route will check for the existence of `user` and `pass` inputs, turning a user's specified password into an MD5 hash.

```
app.post('/', function (req, res) {
  if (req.body && req.body.user && req.body.pass) {
    var hash = crypto
      .createHash("md5")
      .update(req.body.pass)
      .digest('hex');

    userStore[req.body.user] = hash;
    res.send('Thanks for registering ' + req.body.user);
    console.log(userStore);
  }
});
```

When we use our form to register, the console will output the `userStore` object, containing all registered user names and password hashes.

How it works...

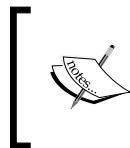
The password hashing portion of this recipe is:

```
var hash = crypto
  .createHash("md5")
  .update(req.body.pass)
  .digest('hex');
```

We've used the dot notation to chain some `crypto` methods together.

First, we create a vanilla MD5 hash with `createHash` (see the *There's more ...* section on how to create unique hashes). We could alternatively create a (stronger) SHA1 hash by passing `sha1` as the argument. The same goes for any other encryption method supported by Node's bundled `openssl` version (0.9.8r as of Node 0.6.17).

For a comparison of different hash functions, see http://hash.iain.tugraz.at/wiki/The_Hash_Function_Zoo.



This site labels certain hash functions as broken, which means a weakness point has been found and published. However, the effort required to exploit such a weakness will often far exceed the value of the data we are protecting.

Then we call `update` to feed our user's password to the initial hash.

Finally, we call the `digest` method, which returns a completed password hash. Without any arguments, `digest` returns the hash in binary format. We pass `hex` (base 16 numerical representation format of binary data, see <http://en.wikipedia.org/wiki/Hexadecimal>) to make it more readable on the console.

There's more...

The `crypto` module offers some more advanced hashing methods for creating even stronger passwords.

Uniquifying hashes with HMAC

HMAC stands for **Hash-based Message Authentication Code**. This is a hash with a secret key (authentication code).

To convert our recipe to using HMAC, we change our `crypto` portion to:

```
var hash = crypto
    .createHmac('md5', 'Super Secret Key')
    .update(req.body.pass)
    .digest('hex');
```

Using HMAC protects us from the use of rainbow tables (pre-computed hashes from a large list of probable passwords). The secret key mutates our hash, rendering a rainbow table impotent (unless an attacker discovers our secret key, for instance, by somehow gaining root access to our server's operating system, at which point rainbow tables wouldn't be necessary anyway).

Hardened hashing with PBKDF2

PBKDF2 is the second version of Password-Based Key Derivation Function, which is part of the Password-Based Cryptographic standard.

A powerful quality of PBKDF2 is that it generates hashes of hashes, thousands of times over. Iterating over the hash multiple times strengthens the encryption, exponentially increasing the amount of possible outcomes resulting from an initial value to the extent that the hardware required to generate or store all possible hashes becomes infeasible.

pbkdf2 requires four components: the desired password, a salt value, the desired amount of iterations, and a specified length of the resulting hash.

A salt is similar in concept to the secret key in our HMAC in that it mixes in with our hash to create a different hash. However, the purpose of a salt differs. A salt simply adds a uniqueness to the hash and it doesn't need to be protected as a secret. A strong approach is to make each salt unique to the hash being generated, storing it alongside the hash. If each hash in a database is generated from a different salt, an attacker is forced to generate a rainbow table for each hash based on its salt rather than the entire database. With PBKDF2, thanks to our salt, we have unique hashes of unique hashes which adds even more complexity for a potential attacker.

For a strong salt, we'll use the `randomBytes` method of `crypto` to generate 128 bytes of random data, which we will then pass through the `pbkdf2` method with the user-supplied password 7,000 times, finally creating a hash 256 bytes in length.

To achieve this, let's modify our POST route from the recipe.

```
app.post('/', function (req, res) {
  if (req.body && req.body.user && req.body.pass) {
    crypto.randomBytes(128, function (err, salt) {
      if (err) { throw err; }
      salt = new Buffer(salt).toString('hex');
      crypto.pbkdf2(req.body.pass, salt, 7000, 256, function (err,
      hash) {
        if (err) { throw err; }
        userStore[req.body.user] = {salt : salt,
          hash : (new Buffer(hash).toString('hex')) };
        res.send('Thanks for registering ' + req.body.user);
        console.log(userStore);
      });
    });
  }
});
```

`randomBytes` and `pbkdf2` are asynchronous, which is helpful because it allows us to perform other tasks or improve the user experience by immediately taking them to a new page while their credentials are being encrypted. This is done by simply placing `res.send` outside of the callbacks (we haven't done this here but it could be a good idea since encryption of this magnitude could take around a second to calculate).

Once we have both our hash and salt values we place them into our `userStore` object. To implement a corresponding login we would simply compute the hash in the same way using that user's stored salt.

We chose to iterate 7,000 times. When PBKDF2 was standardized the recommended iteration count was 1,000. However, we need more iterations to account for technology advancements and reductions in the cost of equipment.

See also

- ▶ *Implementing Digest Authentication* discussed in this chapter
- ▶ *Setting up an HTTPS web server* discussed in this chapter
- ▶ *Generating Express scaffolding* discussed in *Chapter 6, Accelerating Development with Express*

Implementing Digest Authentication

Digest Authentication combines Basic Authentication with MD5 encryption, thus avoiding the transmission of plain text passwords, making for a more secure login method over plain HTTP.

On its own, Digest Authentication is still insecure without an SSL/TLS-secured HTTPS connection. Anything over plain HTTP is vulnerable to man in the middle attacks, where an adversary can intercept requests and forge responses. An attacker could masquerade as the server, replacing the expected Digest response with a Basic Authentication response, thus gaining the password in plain text.

Nevertheless, in the absence of SSL/TLS, Digest Authentication at least affords us some defense in the area of plain text passwords requiring more advanced circumvention techniques.

So in this recipe, we will create a Digest Authentication server.

Getting ready

To begin with, we simply have to create a new folder with a new `server.js` file.

How to do it...

As in the Basic Authentication recipe we create an HTTP server, we'll also be using the `crypto` module to handle the MD5 hashing:

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

var username = 'dave',
    password = 'digestthis!',
    realm = "Node Cookbook",
    opaque;
```

```

function md5(msg) {
    return crypto.createHash('md5').update(msg).digest('hex');
}

opaque = md5(realM);

```

We've made an `md5` function as a shorthand interface to the `crypto` hash methods. The `opaque` variable is a necessary part of the `Digest` standard. It's simply an MD5 hash of `realm` (as also used in Basic Authentication). The client returns the `opaque` value back to the server for an extra means of validating responses.

Now we'll create two extra helper functions, one for authentication and one to parse the `Authorization` header as follows:

```

function authenticate(res) {
    res.writeHead(401, {'WWW-Authenticate': 'Digest realm=' + realm +
        ',qop="auth",nonce=' + Math.random() + ''
        ,opaque=' + opaque + '')});

    res.end('Authorization required.');
}

function parseAuth(auth) {
    var authObj = {};
    auth.split(',').forEach(function(pairs) {
        pair = pairs.split('=');
        authObj[pair[0]] = pair[1].replace(/\//g, '');
    });
    return authObj;
}

```

Finally, we implement the server as shown in the following code:

```

http.createServer(function (req, res) {
    var auth, user, digest = {};

    if (!req.headers.authorization) {
        authenticate(res);
        return;
    }
    auth = req.headers.authorization.replace(/^Digest /, '');
    auth = parseAuth(auth); // object containing digest headers from
    client
    // don't waste resources generating MD5 if username is wrong
    if (auth.username !== username) { authenticate(res); return; }

```

```
digest.ha1 = md5(auth.username + ':' + realm + ':' + password);
digest.ha2 = md5(req.method + ':' + auth.uri);
digest.response = md5([
    digest.ha1,
    auth.nonce, auth.nc, auth.cnonce, auth.qop,
    digest.ha2
].join(':'));

if (auth.response !== digest.response) { authenticate(res); return;
}
res.end('You made it!');

}).listen(8080);
```

Within the browser, this will look exactly the same as Basic Authentication, which is unfortunate as a clear difference between Digest and Basic dialogs could alert the user to a potential attack.

How it works...

When the server sends the `WWW-Authenticate` header to the browser, several attributes are included, consisting of: `realm`, `qop`, `nonce`, and `opaque`.

`realm` is the same as Basic Authentication, and `opaque` is an MD5 hash of the `realm`

`qop` stands for Quality of Protection and is set to `auth`. `qop` can also be set to `auth-int` or simply be omitted. By setting it to `auth`, we cause the browser to compute a more secure final MD5 hash. `auth-int` is stronger still, but browser support for it is minimal.

`nonce` is a similar concept to a salt, it causes the final MD5 hash to be less predictable from an attacker's perspective.

When the user submits their login details via the browser's authentication dialog, an `Authorization` header is returned containing all of the attributes sent from the server, plus the `username`, `uri`, `nc`, `cnonce`, and `response` attributes.

`username` is the user's specified alias, `uri` is the path being accessed (we could use this to secure on a route by route basis), `nc` is a serial counter which is incremented on each authentication attempt, `cnonce` is the browser's own generated `nonce` value, and `response` is the final computed hash.

In order to confirm an authenticated user, our server must match the value of `response`. To do so, it removes the `Digest` string (including the proceeding space) and then passes what remains of the `Authorization` header to the `parseAuth` function. `parseAuth` converts all the attributes into a handy object, and loads it back into our `auth` variable.

The first thing we do with `auth` is check that the username is correct. If we have no match, we ask for authentication again. This could save our server from some unnecessary heavy lifting with MD5 hashing.

The final computed MD5 hash is made from the combination of two previously encrypted MD5 hashes along with the server's `nonce` and `qop` values and the client's `cnonce` and `nc` values.

We called the first hash `digest.ha1`. It contains a colon (:) delimited string of the `username`, `realm` and `password` values. `digest.ha2` is the `request` method (GET) and the `uri` attribute, again delimited by a colon.

The final `digest.response` property has to match `auth.response` which is generated by the browser, so the ordering and specific elements must be precise. To create our `digest.response` we combine `digest.ha1`, the `nonce`, `nc`, the `cnonce`, `qop`, and `digest.ha2` each separated by a colon. For easy reading we put these values into an array running JavaScript's `join` method on them to generate the final string, which is passed to our `md5` function.

If the given username and password are correct, and we've generated `digest.response` correctly, it should match the browser's `response` header attribute (`auth.response`). If it doesn't, the user will be presented with another authentication dialog. If it does, we reach the final `res.end`. We made it!

There's more...

Let's tackle the logout problem.

Logging out of authenticated areas

There is little to no support in browsers for any official logging out method under Basic or Digest Authentication, except for closing the entire browser.

However, we can force the browser to essentially lose its session by changing the `realm` attribute in the `WWW-Authenticate` header.

In a multiuser situation, if we change our global `realm` variable it will cause all users to log out (if there was more than one). So if a user wishes to log out, we have to assign them a unique `realm` that will cause only their session to quit.

To simulate multiple users, we'll remove our `username` and `password` variables, replacing them with a `users` object:

```
var users = {
    'dave' : {password : 'digestthis!'},
    'bob' : {password : 'MyNameIsBob:-D'},
},
realm = "Node Cookbook",
opaque;
```

Our sub-objects (currently containing `password`) will potentially gain three extra properties: `uRealm`, `uOpaque`, and `forceLogOut`.

Next, we'll modify our `authenticate` function as follows:

```
function authenticate(res, username) {
  var uRealm = realm, uOpaque = opaque;
  if (username) {
    uRealm = users[username].uRealm;
    uOpaque = users[username].uOpaque;
  }
  res.writeHead(401, {'WWWAuthenticate':
    'Digest realm=' + uRealm + '"'
    + ',qop="auth",nonce=' + Math.random() + '"'
    + ',opaque=' + uOpaque + '"'});
  res.end('Authorization required.');
}
```

We've added an optional `username` parameter to the `authenticate` function. If `username` is present, we load the unique `realm` and corresponding `opaque` values for that user, sending them in the header.

Inside our server callback we replace this code:

```
// don't waste resources generating MD5 if username is wrong
if (auth.username !== username) { authenticate(res); return; }
```

With the following code:

```
// don't waste resources generating MD5 if username is wrong
if (!users[auth.username]) { authenticate(res); return; }

if (req.url === '/logout') {
  users[auth.username].uRealm = realm + '[' + Math.random() + ']';
  users[auth.username].uOpaque = md5(users[auth.username].uRealm);
  users[auth.username].forceLogOut = true;
  res.writeHead(302, {'Location': '/'});
  res.end();
  return;
}

if (users[auth.username].forceLogOut) {
  delete users[auth.username].forceLogOut;
  authenticate(res, auth.username);
}
```

We check whether the specified username exists inside our `users` object, saving us from further processing if it doesn't. Providing the user is valid, we check the route (we'll be supplying a logout link to the user). If the `/logout` route has been hit, we set up a `uRealm` property on the logged in user's object and a corresponding `uOpaque` property containing an MD5 hash of `uRealm`. We also add a `forceLogOut` Boolean property, setting it to `true`. Then we redirect the user away from the `/logout` to `/`.

The redirect triggers another request, upon which the server detects the presence of our `forceLogOut` property for the currently authenticated user. `forceLogOut` is then removed from the `users` sub-object to prevent it from getting in the way later. Lastly, we pass control over to the `authenticate` function with the special `username` parameter.

Consequently, `authenticate` includes the user-linked `uRealmand` `uOpaque` values in the `WWWAuthenticate` header, breaking the session.

To finish off, we make a few more trivial adjustments.

`digest.ha1` requires the `password` and `realm` values, so it's updated as follows:

```
digest.ha1 = md5(auth.username + ':'  
+ (users[auth.username].uRealm || realm) + ':'  
+ users[auth.username].password);
```

The `password` value is fed in via our new `users` object, and the `realm` value is chosen based upon whether our logged-in user has unique `realm` (a `uRealm` property) set or not.

We change the last segment of our server's code to the following:

```
if (auth.response !== digest.response) {  
  users[auth.username].uRealm = realm + '[' + Math.random() + ']';  
  users[auth.username].uOpaque = md5(users[auth.username].uRealm);  
  authenticate(res, (users[auth.username].uRealm && auth.username));  
  return;  
}  
res.writeHead(200, {'Content-Type': 'text/html'});  
res.end('You made it! <br> <a href="logout"> [ logout ] </a>');
```

Notice the inclusion of a logout link, the final piece.

New `uRealmand` `uOpaque` attributes are generated if the hashes don't match. This prevents an eternal loop between the browser and server. Without this, when we log in as a valid user and then log out, we'd be presented with another login dialog. If we enter a non-existent user, the new login attempt is rejected by the server as normal. However, the browser attempts to be helpful and falls back to the old authentication details with our first logged-in user and original `realm`. But, when the server receives the old login details, it matches the user to their unique `realm`, demanding authentication for `uRealm`, not `realm`. The browser sees the `uRealm` value and matches our non-existent user back to it, attempting to authenticate again as that user, thus repeating the cycle.

By setting a new realm, we break the cycle because an extra realm is introduced which the browser has no record of, so it defers to the user by asking for input.

See also

- ▶ *Implementing Basic Authentication* discussed in this chapter
- ▶ *Cryptographic password hashing* discussed in this chapter
- ▶ *Setting up an HTTPS web server* discussed in this chapter

Setting up an HTTPS web server

For the most part, HTTPS is the solution to many of the security vulnerabilities such as (network sniffing, and man in the middle) faced over HTTP.

Thanks to the core `https` module. It's really simple to set up.

Getting ready

The greater challenge could be in actually obtaining the necessary SSL/TLS certificate.

In order to acquire a certificate, we must generate an encrypted Private Key, and from that we generate a Certificate Signing Request. This is then passed onto a Certificate Authority (a commercial entity specifically trusted by browser vendors—naturally this means we have to pay for it). Alternatively, the CA may generate your Private Key and Certificate Signing Request on your behalf.



The company, StartSSL provides free certificates. An article about using StartSSL certificates with Node can be found at <https://www.totallnate.net/setting-up-free-ssl-on-your-node-server>.

After a verification process, the Certificate Authority (CA) will issue a Public Certificate enabling us to encrypt our connections.

We can short cut this process and authorize our own certificate (self-sign), naming ourselves as the CA. Unfortunately, if the CA isn't known to a browser, it will vividly warn the user that our site isn't to be trusted and that they may be under attack. This isn't so good for positive brand image. So while we may self-sign during development, we would most likely need a trusted CA for production.

For development, we can quickly use the `openssl` executable (available by default on Linux and Mac OS X, and we can obtain a Windows version from `http://www.openssl.org/grelated/binaries.html`) to generate necessary Private Key and Public Certificate:

```
openssl req -new -newkey rsa:1024 -nodes -subj '/O=Node Cookbook'  
-keyout key.pem -out csr.pem && openssl x509 -req -in csr.pem -signkey  
key.pem -out cert.pem
```

This executes `openssl` twice on the command line: once to generate basic Private Key and Certificate Signing Request, and again to self-sign Private Key, thus generating Certificate (`cert.pem`).

In a real production scenario, our `-subj` flag would hold more details, and we would want to acquire our `cert.pem` file from a legitimate CA. But this is fine for private, development, and testing purposes.

Now that we have our key and certificate, we simply need to make our server, so we'll create a new `server.js` file.

How to do it...

Within `server.js` we write the following code:

```
var https = require('https');  
var fs = require('fs');  
  
var opts = {key: fs.readFileSync('key.pem'),  
cert: fs.readFileSync('cert.pem')};  
  
https.createServer(opts, function (req, res) {  
    res.end('secured!');  
}).listen(4443); //443 for prod
```

And that's it!

How it works...

The `https` module depends upon the `http` and `tls` modules, which in turn rely upon the `net` and `crypto` modules. SSL/TLS is transport layer encryption, meaning that it works at a level beneath HTTP, at the TCP level. The `tls` and `net` modules work together to provide an SSL/TLS-encrypted TCP connection, with HTTPS layered on top of this.

When a client connects via HTTPS (in our case, at the address `https://localhost:4443`), it attempts a TLS/SSL handshake with our server. The `https` module uses the `tls` module to respond to the handshake in a series of fact-finding interchanges between the browser and server. (For example, what SSL/TLS version do you support? What encryption method do you want to use? Can I have your public key?)

At the end of this initial interchange, the client and server have an agreed shared secret. This secret is used to encrypt and decrypt content sent between the two parties. This is where the `crypto` module kicks in, providing all of the data encryption and decryption functionality.

For us, it's as simple as requiring the `https` module, providing our certificates, then using it just like we would an `http` server.

There's more...

Let's see a few HTTPS use cases.

HTTPS in Express

Enabling HTTPS in Express is just as simple:

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

var opts = {key: fs.readFileSync('key.pem'),
           cert: fs.readFileSync('cert.pem')};

var app = express.createServer(opts).listen(8080);

app.get('/', function (req, res) {
  res.send('secured!');
});
```

Securing Basic Authentication with SSL/TLS

We can build anything into our `https` server that we could into an `http` server. To enable HTTPS in our Basic Authentication recipe all we do is alter:

```
https.createServer(function (req, res) {
```

To the following:

```
var opts = {key: fs.readFileSync('key.pem'),
           cert: fs.readFileSync('cert.pem')};

https.createServer(opts, function (req, res) {
```

See also

- ▶ *Cryptographic password hashing* discussed in this chapter
- ▶ *Implementing Basic Authentication* discussed in this chapter

Preventing cross-site request forgery

There's a problem with every browser's security model that, as developers, we must be aware of.

When a user has logged in to a site, any requests made via the authenticated browser are treated as legitimate—even if the links for these requests come from an email, or are performed in another window. Once the browser has a session, all windows can access that session.

This means an attacker can manipulate a user's actions on a site they are logged in to with a specifically crafted link, or with automatic AJAX calls requiring no user interaction except to be on the page containing the malicious AJAX.

For instance, if a banking web app hasn't been properly CSRF secured, an attacker could convince the user to visit another website while logged in to their online banking. This website could then run a POST request to transfer money from the victim's account to the attacker's account without the victim's consent or knowledge.

This is known as a **Cross - Site Request Forgery (CSRF)** attack. In this recipe, we'll be implementing a secure HTML login system with CSRF protection.

Getting ready

We'll be securing our Profiler Web App from the *Making an Express Web App* recipe discussed in *Chapter 6, Accelerating Development with Express*. We'll want to get a hold of our `profiler` app, with the `profiler/app.js` and `profiler/login/app.js` files open and ready for editing.

Without SSL/TLS encryption, HTML-based logins are subject to at least the same vulnerabilities as Basic Authorization. So for basic security, we'll be adding HTTPS to our app. So we need `cert.pem` and `key.pem` files from the previous recipe.

We'll also need to have MongoDB running with our stored user data from recipes in *Chapter 6, Accelerating Development with Express*, since our `profiler` app relies upon it.

```
sudo mongod --dbpath [PATH TO DB]
```

How to do it...

First, let's secure our entire app with SSL, the top of `profiler/app.js` should look like the following code:

```
var express = require('express')
, routes = require('./routes')
, fs = require('fs');

var opts = {key: fs.readFileSync('key.pem'),
cert: fs.readFileSync('cert.pem')};

var app = module.exports = express.createServer(opts);
```

The `admin` section of `profiler` is where a CSRF attack could take place, so let's open up `profiler/login/app.js` and add the `express.csrf` middleware. The top of the `app.configure` callback in `profiler/login/app.js` should look like the following code:

```
app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');

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

  app.use(express.cookieParser());
  app.use(express.session({secret: 'kooBkooCedoN'}));

  app.use(express.csrf());
  //rest of configure callback
});
```

Express 3.x.x

Don't forget, in Express 3.x.x the secret goes as a string into `cookieParser` instead of an object into `session:express.cookieParser('kooBkooCedoN');`.

The `csrf` middleware is dependent upon the `bodyParser` and `session` middleware, so it must be placed below these.

Now if we navigate to `https://localhost:3000/admin` and attempt to log in (dave, expressrocks), we will receive a 403 Forbidden response, even though we use correct details.

That's because our login app is now looking for an additional POST parameter called `_csrf` in all of our POST forms, which must match the `_csrf` value stored in the user's session object.

Our views need to know the value of `_csrf` so it can be placed in our forms as a hidden element.

We'll use `dynamichelper` to supply `req.session._csrf` to all login views.

```
app.dynamichelpers({ //Express 3.x.x would use app.locals.use instead
  user: function (req, res) { //see Chapter 6 for details.
    return req.session.user;
  },
  flash: function (req, res) {
    return req.flash();
  },
  _csrf: function (req) {
    return req.session._csrf;
  }
});
```

Next, we'll create a view called `csrf.jade` in the `login/views` folder as follows:

```
input(type="hidden", name="_csrf", value=_csrf);
```

Now we include `csrf.jade` in each of our POST forms.

`login.jade`:

```
// prior login jade code above
if user
  form(method='post')
    input(name="_method", type="hidden", value="DELETE")
    include csrf
    p Hello #{user.name}!
    a(href='javascript:', onClick='forms[0].submit()') [logout]

  include admin
  include ../../views/profiles

else
  p Please log in
  form(method="post")
    include csrf
    fieldset
      legend Login
    //rest of login.jade
```

addfr m jade:

```
fields = ['Name', 'Irc', 'Twitter', 'Github', 'Location',  
'Description'];  
form#addfrm(method='post', action='/admin/add')  
  include csrf  
  fieldset  
    legend Add  
  //rest of addfrm.jade
```



Updating and maintaining a site with many different POST forms could pose as challenging. We would have to manually alter every single form. See how we can auto-generate CSRF values for all forms in the *There's More...* section.

Now we can log in, add profiles, and log out without a 403 Forbidden response.

However, our / deI route is still susceptible to CSRF. The GET requests are not typically supposed to trigger any changes on the server. They are intended simply to retrieve information. However, like many other apps in the wild, the developers (that's us) were lazy when they built this particular functionality and decided to coerce a GET request to do their bidding.

We could turn this into a POST request and then secure it with CSRF, but what about an app that has hundreds of these deviant GETs?

Let's find out how to protect our / del route. In `login/routes/index.js` add the following code:

```
exports.delprof = function (req, res) {
  if (req.query._csrf !== req.session._csrf) {
    res.send(403);
    return;
  };
  profiles.del(req.query.id, function (err) {
    if (err) { console.log(err); }
    profiles.pull(req.query.p, function (err, profiles) {
      req.app.helpers({profiles: profiles});
      res.redirect(req.header('Referrer') || '/');
    });
  });
};
```

Our changes make it so we can't delete profiles until we include `_csrf` in the query string, so in `views/admin.jade`:

```
mixin del(id)
  td
    a.del(href='/admin/del?id=#{{id}}&p=#{{page}}&_csrf=#{{_csrf}}')
      &#10754;
  // rest of admin.jade
```

How it works...

The `csrf` middleware generates a unique token which is held in the user's session. This token must be included in any actioning requests (logging in, logging out, adding or deleting) as an attribute named `_csrf`.

If the `_csrf` value in the request body (or query string for GET) doesn't match the `_csrf` token stored in the `session` object, the server denies access to that route and therefore prevents the action from occurring.

How does this prevent a CSRF attack? In a plain CSRF exploit, the attacker has no way of knowing what the `_csrf` value is, so they are unable to forge the necessary POST request.

Our `/del` route protection is less secure. It exposes the `_csrf` value in the address, potentially creating a very small, but nonetheless plausible, window of opportunity for an attacker to grab the `_csrf` value. This is why it's best for us to stick with the POST/DELETE/PUT requests for all action-related endeavors, leaving GET requests for simple retrieval.

Cross-site scripting (XSS) circumvention

This protection is rendered moot in the event of an accompanied XSS exploit, whereby an attacker is able to implant their own JavaScript within the site (for example, through exploiting an input vulnerability). JavaScript can read any elements in the page it resides on, and view non-`HttpOnly` cookies with `document.cookie`.

There's more...

We'll take a look at a way to automatically generate CSRF tokens for login forms, but we should also bear in mind that CSRF protection is only as good as our ability to code tightly.

Auto-securing the POST forms with the CSRF elements

Ensuring that all the POST forms in our app contain a hidden `_csrf` input element could be an arduous task on a site of any significant scale.

We can interact directly with some Jade internals to automatically include these elements.

First, in `login/app.js` we add the following setting to our configuration:

```
app.set('view options', {compiler: require('./  
customJadeCompiler')});
```

Express allows us to push particular options to whatever view engine we are using. One of the Jade options (our view engine) is `compile`, which allows us to define our own custom Jade interpreter.

Let's create `customJadeCompiler.js` placing it in the `login` directory.

First, we'll require some modules and set up our new compiler class as follows:

```
var jade = require('jade');  
var util = require('util');  
  
// inherit from Jades Compiler  
var CompileWithCsrf = function (node, options) {  
    jade.Compiler.call(this, node, options);  
};
```

Next we use `util.inherits` to inherit our new compiler's prototype from the Jade's compiler.

```
// inherit from the prototype  
util.inherits(CompileWithCsrf, jade.Compiler);
```

Then we modify the Jade's internal `visitTag` method (which we've inherited from `jade.Compiler`):

```
CompileWithCsrf.prototype.visitTag = function (tag) {  
  
    if (tag.name === 'form' && tag.getAttribute('method').match(/  
post/i)) {  
  
        var csrfInput = new jade.nodes.Tag('input')  
            .setAttribute('type', '"hidden"')  
            .setAttribute('name', '"csrf"')  
            .setAttribute('value', '_csrf');  
  
        tag.block.push(csrfInput);  
  
    }  
    jade.Compiler.prototype.visitTag.call(this, tag);  
};
```

Finally, we load our new compiler into `module.exports` so it's passed via `require` to the `compiler` option of the `view options` setting in `app.js`:

```
module.exports = CompileWithCsrf;
```

We create a new class-type function, applying the `call` method to `jade.Compiler`. When we pass the `this` object to the `call` method, we essentially inherit the main functionality of `jade.Compiler` into our own `CompileWithCsrf` class-type function. It's a great way to re-use code.

However, `jade.Compiler` also has a modified prototype which must be incorporated into our `CompileWithCsrf` in order to fully mimic `jade.Compiler`.

We used `util.inherits`, but we could instead have said:

```
CompileWithCsrf.prototype = new jade.Compiler();
```

Or:

```
CompileWithCsrf = Object.create(jade.Compiler);
```

Or even:

```
CompileWithCsrf.prototype.__proto__ = jade.Compiler.prototype;
```

`Object.create` is the Ecmascript5 way, `new` is the old way and `__proto__` is the non-standard way that should probably be avoided. These all inherit the additional methods and properties of `jade.Compiler`. However, `util.inherits` is preferred because it also adds a special `super` property containing the object we originally inherited from.

Using `call` and `util.inherits` essentially allows us to clone the `jade.Compiler` object as `CompileWithCsrf`, which means we can modify it without touching `jade.Compiler` and then allow it to operate in place of `jade.Compiler`.

We modify the `visitTag` method, which processes each tag (for example, `p`, `div`, and so on) in a Jade view. Then we look for the `formtags` with methods set to `post`, using a regular expression since the `method` attribute may be in upper or lower case, being wrapped in double or single quotes.

If we find a `form` with POST formatting, we use the `jade.Nodes` constructor to create a new input node (a Jade construct, in this case rolling as an HTML element), which we then call `setAttribute` (an internal Jade method) on three times to set the `type`, `name` and `value` fields. Notice `name` is set to `"_csrf"` but `value` contains `'_csrf'`. The inner double quotes tell Jade we intend a string. Without them, it treats the second parameter as a variable, which is exactly what we want in the case of `value`. The `value` attribute is therefore rendered according to `_csrf` dynamic Helper defined in `app.js` (which is likewise taken from `req.session._csrf` as generated by the `express.csrf` middleware).

Now that our `_csrf` tokens are automatically included in every POST form, we can remove the `csrf.jade` includes from the `login.jade` and `addfrom.jade` views.

Eliminating cross-site scripting (XSS) vulnerabilities

Cross-site scripting attacks are generally preventable, all we have to do is ensure any user input is validated and encoded. The tricky part comes where we improperly or insufficiently encode user input.

When we take user input, much of the time we'll be outputting it to the browser at a later stage, this means we must embed it within our HTML.

XSS attacks are all about breaking context. For instance, imagine we had some Jade that links to a user profile by their username:

```
a ( href=username) !{username}
```

This code is exploitable in two ways. First, we used `!{username}` instead of `#{}{username}` for the text portion of our anchor link element. In Jade `#{}` interpolation escapes any HTML in the given variable. So if an attacker was able to insert:

```
<script>alert('This is where you get hacked!')</script>
```

As their `username`, `#{}{username}` would render:

```
&t;script&gt;alert('This is where you get hacked!')&t;/script&gt;
```

Whereas, `!{username}` would be unescaped (for example, HTML would not be replaced by escape characters like `&t;` in place of `<`). The attacking code could be changed from an innocent (though jaunty) `alert` message, to successfully initiated Forged Requests, and our CSRF protection would be futile since the attack is operating from the same page (JavaScript has access to all data on the page, and the attacker has gained access to our page's JavaScript via XSS).

Jade HTML-escapes variables by default, which is a good thing. However, proper escaping must be context aware, and simply converting HTML syntax into its corresponding entity codes is not enough.

The other vulnerable area in our bad Jade code is the `href` attribute. Attributes are a different context to simple nested HTML. Unquoted attributes are particularly susceptible to attack, for instance, consider the following code:

```
<a href=profile>some text</a>
```

If we could set `profile` to `profile onClick=javascript:alert('gotcha')`, our HTML would read:

```
<a href=profile onClick=javascript:alert('gotcha')>some text</a>
```

Again, Jade partially protects us in this sense by automatically quoting variables inserted to attributes. However, our vulnerable attribute is the `href` attribute, which is another sub context of the URL variety. Since it isn't prefixed with anything, an attacker might input their username as `javascript:alert('oh oh!')` so the output of:

```
a ( href=userName) !{userName}
```

Would be:

```
<a href="javascript:alert('oh oh!')"> javascript:alert('oh oh!') </a>
```

The `javascript:` protocol allows us to execute JavaScript at the link level, allowing a CSRF attack to be launched when an unsuspecting user clicks a malicious link.

 These trivial examples are elementary. XSS attacks can be much more complex and sophisticated. However, we can follow the Open Web Application Security Projects 8 input sanitizing rules that provide extensive protection against XSS:

[https://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)

Validator module

Once we understand how to clean user input, we could use regular expressions to quickly apply specific validation and sanitization methods. However, for a simpler life, we could also use the third-party `validator` module which can be installed with npm Documentation is available on the Github page: <https://github.com/chriso/node-validator>.

See also

- ▶ *Setting up an HTTPS web server* discussed in this chapter
- ▶ *Initializing and using a session* discussed in *Chapter 6, Accelerating Development with Express*
- ▶ *Making an Express web app* discussed in *Chapter 6, Accelerating Development with Express*

8

Integrating Network Paradigms

In this chapter we will cover:

- ▶ Sending email
- ▶ Sending SMS
- ▶ Communicating with TCP
- ▶ Creating an SMTP server
- ▶ Implementing a virtual hosting paradigm

Introduction

Node's capabilities extend far beyond simply serving web pages. Node's core focus of supporting various computing tasks and networking objectives with a simple to understand interface allows developers like ourselves to unleash our creativity and innovate increasingly interconnected solutions and ideas.

In this chapter, we'll be looking at some fundamental examples of such interconnectivity with the knowledge that we can take these archetypes, growing and combining them into larger applications.

Knowledge of how to implement network paradigms can help us to exceed the normal boundaries of a web app, providing advanced functionality to our users, and implementing more ways for individuals to connect with our services.

Sending email

In many platforms, the ability to send email comes standard, but the Node approach leaves the email functionality to the developer.

Thankfully, there are some excellent module creators in the Node community who have already created modules for sending email. In this recipe, we'll be using the well-featured third-party `nodemailer` module to send an imaginary newsletter to a list of recipients.

Getting ready

In order to send an email, we'll need a functioning SMTP server that we can connect to. In a later recipe, we'll be creating our own SMTP server, but for now we'll have to acquire some details of our SMTP to use our client with.

If we have an email address, we have access to an SMTP server. We can find out the SMTP host address from our provider.

If required we can obtain access to an SMTP server by signing up for a Gmail account (at `mail.google.com`). Once we have an account, we can use `smtplib.gmail.com` as the host, with our Gmail address as the username.

We'll create a new folder with a new file called `mailout.js` to hold our code.

How to do it...

There are three main elements to using `nodemailer`. They are as follows:

1. Setting up the SMTP transport
2. Putting together the message object (which includes the transport)
3. Passing the object to the `sendMail` method.

Let's add the `nodemailer` module and create the transport as shown in the following code:

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

var transport = nodemailer.createTransport('SMTP', {
  host: 'smtp.gmail.com',
  secureConnection: true,
  port: 465,
  auth: {
    user: "ourGmailAddress@gmail.com",
    pass: "ourPassword"
  }
});
```

We will need to fill in our own SMTP settings for the `user` and `pass` values.

We've used the `secureConnection` setting and set the port to 465, so we can use Gmail's SSL/TLS-secured SMTP server.

Now we incorporate our configured transport into an object which we'll call `msg` as follows:

```
var msg = {
  transport: transport,
  text:      "Hello! This is your newsletter, :D",
  from:      "Definitely Not Spammers <spamtot@ok.com>",
  subject:   "Your Newsletter"
};
```

Notice we haven't set a `to` property on the object. We're going to mail out to multiple addresses, so `to` will be set dynamically. For testing purposes, we'll create an array of `mailinator` email addresses. **Mailinator** (`http://www.mailinator.com`) is a free service that allows us to quickly create temporary email addresses simply by sending an email to an invented address.

```
var maillist = [
  'M One <mailto:1@mailinator.com>',
  'M Two <mailto:2@mailinator.com>',
  'M Three <mailto:3@mailinator.com>',
  'M Four <mailto:4@mailinator.com>',
  'M Five <mailto:5@mailinator.com>'
];
```

Now we simply loop through and send our newsletter to each address as follows:

```
var i = 0;
maillist.forEach(function (to) {
  msg.to = to;
  nodemailer.sendMail(msg, function (err) {
    if (err) { console.log('Sending to ' + to + ' failed: ' + err); }
    console.log('Sent to ' + to);

    i += 1
    if (i === maillist.length) { msg.transport.close(); }
  });
});
```

If we point our browser at `http://mailto:1@mailinator.com` (or `mailto:2, mailto:3`, and so on), we should see our message in Mailinator's temporary inbox.

How it works...

With Nodemailer's `createTransport` method, we can quickly configure our app with the required SMTP settings, later including these settings in the `msg` object as used by the `sendMail` method.

We don't set an initial `to` property because it's modified through each iteration of `mailList.forEach` before being passed into the `sendMail` method.

`sendMail` is asynchronous, as most methods with callbacks are (`forEach` being an exception). After each `sendMail` is called, `forEach` moves on and calls the next `sendMail` without waiting for the `sendMail` invocation to complete. This means that the `forEach` loop will finish before all the `sendMail` calls are finished. So in order to know when all the mail has been sent we use a counter (`i`).

Each time an email is sent we increase our `i` variable by 1. Once `i` is equivalent to the size of our `mailList` array, all emails have been dispatched so we call `transport.close`.

Nodemailer opens multiple connections (a connection pool) to the SMTP server and reuses those connections for all the emails being sent. This ensures fast and efficient emailing, removing the overhead of opening and closing connections for each email sent. `transport.close` shuts down the connection pool and thus allows our app to finish execution.

There's more...

Nodemailer is a well-featured, highly-configurable mailing module as we'll see.

Using sendmail as an alternative transport

Many hosting providers have a `sendmail` service that connects to a default SMTP server, the details of which we need not know. Nodemailer will interface with `sendmail` if we simply alter our `transport` object to:

```
var transport = nodemailer.createTransport("Sendmail");
```

If `sendmail` isn't in our host server's environment PATH variable (to find out simply type `sendmail` from an SSH prompt), we can instead specify where `sendmail` is as follows:

```
var transport = nodemailer.createTransport("Sendmail", "/to/sendmail");
```

HTML emails

Emails can contain HTML and gracefully degrade to plain text for basic user agents. To send an HTML email, we simply add the `html` property to our `msg` object:

```
var msg = {
  // prior properties: transport
  text: "Hello! This is your newsletter, :D",
  html: "<b>Hello!</b><p>This is your newsletter, :D</p>",
  // following properties: from, subject
};
```

The plain text should be included along with HTML, allowing a fallback for email clients with no HTML support.

If we don't want to write the text portion separately, we can have Nodemailer extract the text from the HTML for us, using the `generateTextFromHTML` property as shown in the following code:

```
var msg = {
  transport: transport,
  html: "<b>Hello!</b><p>This is your newsletter, :D</p>",
  createTextFromHTML: true,
  from: "Definitely Not Spammers <spamnot@ok.com>",
  subject: "Your Newsletter"
};
```

Sending attachments

What if we wanted to tell a really bad joke, using email attachments?

We'll dynamically create a text file and load an image file from disk, both of which we'll attach to an email.

For the image, we'll be using `deer.jpg` (which can be found in the supporting code files). This should go in the same folder as our mail out file (let's call it `mail.out_attachments.js`).

```
var nodemailer = require('nodemailer');
var msg = {
  transport: nodemailer.createTransport('SMTP', {
    host: 'smtp.gmail.com',
    secureConnection: true,
    port: 465,
    auth: {
      user: "our Gmail Address@googlemail.com",
      pass: "our Password"
    }
});
```

```
        },
    text: "Answer in the attachment",
    from: "The Attacher <attached@files.com>",
    subject: "What do you call a deer with no eyes?",
    to: "anyemail@anyaddress.com",
    attachments: [
        {fileName: 'deer.txt', contents:'no eye deer.'},
        {fileName: 'deerWithEyes.jpg', filePath: 'deer.jpg'}
    ]
};

nodemailer.sendMail(msg, function (err) {
    if (err) { console.log('Sending to ' + msg.to + ' failed: ' + err); }
    console.log('Sent to ' + msg.to);
    msg.transport.close();
});
```

Of course this is a proof of concept for attachments, and isn't the best use of email. Attachments are provided as an array of objects within the `msg` object. Each attachment object must have a `fileName` property, which is the filename given to the attachment in the email. This doesn't have to match the name of the actual file loaded from disk.

The file contents can be written directly via the `contents` property, using a string or a `Buffer` object, or we can use `filePath` to stream a file from disk (we can also pass a stream directly to a `sourceStream` property).

See also

- ▶ *Sending SMS* discussed in this chapter
- ▶ *Creating an SMTP server* discussed in this chapter

Sending SMS

Being able to send SMS text messages to our users is another way for us to connect with them

It is possible to connect our computer to a GSM modem, interact with specialized libraries (such as Asterisk, `asterisk.org`, combined with `ngSMS`, `ozekisms.com`), and interface with the libraries and the telephony equipment to send SMS messages.

There are easier ways though. Services like Twilio provide gateway SMS services, where we contact them via an HTTP REST API and they handle the SMS sending for us.

In this recipe, we'll convert our newsletter mail out app into a blanket SMS service using the `t wi l i o` module.

Getting ready

This requires a Twilio account (`ht t ps : / / www. t wi l i o . c o m/ t r y - t wi l i o`). Once signed up and logged in we should take note of our Account SID, Auth Token, and Sandbox phone number (we may have to select our country of residence to obtain the appropriate Sandbox number).

We'll need some phone numbers to send texts to for testing purposes. In the sandbox mode (which is what we'll be using for development), any number we want to text or call must go through a verification process. We do this by selecting the **Numbers** link from the **Account** section, and clicking **Verify a Number**. Twilio will then call that number and expect a PIN provided on screen to be entered for confirmation.

Let's create a new file, `s m s out . j s`, and install the `t wi l i o` helper module:

```
npm i n s t a l l t wi l i o
```

How to do it...

First, we require `t wi l i o` and put together some configuration settings:

```
var t wi l i o = r e q u i r e (' t wi l i o ' );
var s e t t i n g s = {
  s i d : ' Ad054bz5be4se5dd211295c38446da2ff d' ,
  t o k e n: ' 3e0345293rhebt45r6er ta89xc89v103' ,
  h o s t n a m e : ' dummyhost ' ,
  p h o n e n u m b e r : ' +14155992671' // s a n d b o x n u m b e r
}
```

Twilio phone number

Before we can start interacting with the Twilio service, we have to specify a registered Twilio phone number in order to create our `p h o n e`. For development purposes, we can simply use the sandbox number, which can be found from the Twilio dashboard (`ht t p : / / www. t wi l i o . c o m/ user / account`). In a production scenario, we would need to upgrade our account and purchase a unique phone number from Twilio.

With our settings present and correct we're ready to create a Twilio client, using it to initialize a virtual phone:

```
var restClient = new (twilio.RestClient)(settings.sid, settings.token);

var client = new (twilio.Client)(settings.sid,
                                settings.token,
                                settings.hostName);

var phone = client.getPhoneNumber(settings.phoneNumber);
```

We created `restClient` here also, which offers API calls reflective of Twilio's raw REST API. We'll be using `restClient` to check the status of our SMS message in order to determine if the message has been sent from Twilio to the target phone.

Now we define a list of numbers to text (we'll have to provide our own) to text, much like our `mailList` array in the previous recipe:

```
var smsList = [
  '+44770xxxxxx1',
  '+44770xxxxxx2',
  '+44770xxxxxx3',
  '+44770xxxxxx4',
  '+44770xxxxxx5'
];
```



Unless we have upgraded our account, any number on `smsList` must be pre-verified with Twilio. This can be done through the Twilio Numbers account section (`https://www.twilio.com/user/account/phone-numbers/`).

Then we simply loop through `smsList` and use `phone` to send an SMS message to each recipient as follows:

```
var msg = 'SMS Ahoy!';
smsList.forEach(function(to) {
  phone.sendSms(to, msg, {}, function(sms) { console.log(sms); });
});
```

This should work fine, except that the process won't exit (because `twilio` initializes a server to receive Twilio callbacks) and we don't know when an SMS is actually sent from Twilio to the recipient. One way to check is to make another request to the Twilio REST API asking for a status update. The `twilio.RestClient` makes this easy for us as follows:

```

phone.sendSms(to, msg, {}, function(sms) {
    restClient.getSmsInstance(sms.smsDetails.sid, function
(presentSms) {
        // process presentSms using it's status property.
    });
});

```

If our SMS hasn't been sent on the first call, we need to wait and check it again. Let's make some final improvements as shown in the following code:

```

var msg = 'SMS Ahoy!', i = 0;
smslist.forEach(function (to) {
    phone.sendSms(to, msg, {}, function (sms) {

        function checkStatus(smsInstance) {
            restClient.getSmsInstance(smsInstance.sid, function (presentSms)
{
                if (presentSms.status === 'sent') {
                    console.log('Sent to ' + presentSms.to);
                } else {
                    if (isNaN(presentSms.status)) {
                        //retry: if its not a number (like 404, 401), it's not an
error
                        setTimeout(checkStatus, 1000, presentSms);
                        return;
                    }
                    //it seems to be a number, let's notify, but carry on
                    console.log('API error: ', presentSms.message);
                }
                i += 1;
                if (i === smslist.length) { process.exit(); }
            });
        };
        checkStatus(sms.smsDetails);
    });
});

```

Now the console will output each time a number has been confirmed as sent. When all numbers have been messaged the process exits.

How it works...

The `twilio.RestClient` gives us access to low-level interactions with the Twitter API via the `twilio` helper. This simply wraps generic API calls with our preset authorization settings for us, making the HTTP requests on our behalf.

`twilio.Client` is a higher-level API presented by the `twilio` helper and handles two-way interactions between Twilio and the client. The third parameter that must be provided to initialize a new client is the `hostname` parameter. The `twilio` module provides this to Twilio as part of a callback URL requested from the Twilio servers to confirm that an SMS message has been sent.

We ignored this behavior and supplied a dummy hostname, implementing our own method of confirming an SMS was sent. Our method doesn't require us to have a live server address accessible from the Web (see the *There's more...* section for a live server implementation that uses the `hostname` property as intended).

We use the `sendSMS` method of our created `phone` object to make an HTTP request of the Twilio API via the `twilio` module, passing in the desired recipient, message, and a callback function.

Once the request is made, our callback is triggered with the initial `sms` object. We use this to determine the ID that Twilio has given our `sendSMS` request with `smsInstance.sid` (which is `sms.smsDetails.sid`).

`smsInstance.sid` is passed to `restClient.getSMSInstance` which provides us with an updated instance of our `smsInstance` object, we call it `presentSMS`. This call is made from within a custom, self-calling function expression called `checkStatus`, which has our initial `sms.smsDetails` object passed to it.

We are looking to see if Twilio has sent our text message yet. If it has, `presentSMS.status` will be `sent`. If it's anything other than this, we want to wait a short while and then ask Twilio for another update on the status of our queued SMS message. That is, unless the returned status is a 404, in which case there has been an issue and we need to notify the user, but continue on to process the next SMS message.

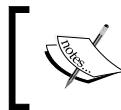
As in the *Sending email* recipe, we keep a count of how many messages are sent. Once they total the amount of recipients we exit the process, since `smsout.js` is a command-line app (in a server scenario we wouldn't need or want to do this).

There's more...

The Twilio module's versatility stretches beyond sending SMS messages. It can also transparently handle Twilio callbacks for us through emitting events.

Using the processed event listener

With a live public server, we would be better off providing our hostname to `twilio.Client` so it can manage callback URL requests.



For this code to work, it must be hosted on a live public server. For more information on hosting Node on live servers, see *Chapter 10, Taking It Live*.

We would remove `restClient` and change our `settings` object to the following:

```
var settings = {
  sid: 'Ad054bz5be4se5dd211295c38446da2ffd',
  token: '3e0345293rhebt45r6ertha89xc89v103',
  hostName: 'nodecookbook.com',
  phoneNumber: '+14155992671' // sandbox number
};
```

Then our SMS sending code is simply:

```
var i = 0;
smsList.forEach(function(to) {
  phone.sendSms(to, msg, {}, function(sms) {
    sms.on('processed', function(req) {
      i += 1;
      console.log('Message to ' + req.To +
        ' processed with status: ' + req.SmsStatus);
      if (i === smsList.length) {process.exit();}
    });
  });
});
```

The `twilio` client provides a `statusCallback` URL to Twilio. Twilio will request this URL (something like `http://nodecookbook.com:31337/automation/0`) to confirm and the `twilio` helper module will emit a `processed` event to notify us of Twilio's confirmation. We listen out for this event, checking the given `SmsStatus` via that `req` object to confirm success.

Making an automated phone call

For this next example to work, we would need to have a valid hostname and be running our app on a web-exposed server as in the previous section.



For this code to work, it must be hosted on a live public server. For more information on hosting Node on live servers, see *Chapter 10, Taking It Live*.

To make a call we start with the usual setup.

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

var settings = {
  sid: 'Ad054bz5be4se5dd211295c38446da2ffd',
  token: '3e0345293rhebt45r6ert a89xc89v103',
  host name: 'nodecookbook.com',
  phonenumer: '+14155992671' // sandbox number
};

var client = new (twilio.Client)(settings.sid,
                                settings.token,
                                settings.host name);

var phone = client.getPhoneNumber(settings.phonenumber);
```

Then we use the `makeCall` method as follows:

```
phone.makeCall('+4477xxxxxx1', {}, function(call) {
  call.on('answered', function(req, res) {
    console.log('answered');
    res.append(new (twilio.Twiml).Say('Meet us in the abandoned
factory'));
    res.append(new (twilio.Twiml).Say('Come alone', {voice:
'woman'}));
    res.send();
  }).on('ended', function(req) {
    console.log('ended', req);
    process.exit();
  })
});
```

If our account is not upgraded, whatever number we supply to `makeCall` must be verified through the Twilio Numbers area in the account section (<https://www.twilio.com/user/account/phone-numbers/>).

`makeCall` invokes a callback with a `call` parameter. `call` is an event emitter, with `answered` and `ended` events. The `twilio` module transparently converts Twilio confirmation callbacks into these events.

The `answered` event feeds the `req` and `res` objects to its callback. `req` holds information about the outgoing call, and `res` has methods which allow us to interact with the call, namely `res.append` and `res.send`.

To send a computerized text-to-speech message to the recipient, we instantiate a new instance of the `twilio` module's `Twiml` class and use the `Say` method (watch out for the unusual convention of giving a capital S to a non-class, using a small s will throw an error).

`TwiML` is Twilio's markup language. It's simply API-specific XML. The `twilio` module provides the `Twiml` class to handle the ugly business of forming the necessary XML. We use it to create two `Say` verbs. Behind the `twilio` scenes, the two `append` calls followed by the `send` invocation would create and issue the following `TwiML` to `twilio`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Response>
  <Say>
    Meet us in the abandoned factory
  </Say>
  <Say voice="woman">
    Come alone
  </Say>
</Response>
```

The `TwiML` code is received by `twilio` and converted into speech. After the woman's voice says `Come alone` the phone call ends, triggering the `ended` event in our app (which the `twilio` module emits as a result of receiving an HTTP request from `twilio` signifying the call has ended).

We listen to the `ended` event determining when `twilio` (or the recipient) have hung up. Once `ended` has triggered we exit the process, outputting the `req` object as an overview of the call.

See also

- ▶ *Sending email* discussed in this chapter
- ▶ *Communicating with TCP* discussed in this chapter
- ▶ *Chapter 10, Taking It Live*

Communicating with TCP

The Transmission Control Protocol (TCP) provides the backbone of HTTP communications. With TCP, we can open up interfaces between processes running on separate server hosts and remotely communicate between processes with less overhead and fewer complexities than HTTP.

Node provides us with the `net` module for creating TCP interfaces. When it comes to scaling, reliability, load balancing, synchronization, or real-time social communications, TCP is a fundamental element.

In this recipe, we're going to demonstrate the sort of foundation needed to communicate between processes over a network by setting up a TCP connection that allows us to remotely monitor and filter HTTP headers of website hits in real time.

Getting ready

We'll need two new files: `server.js` and `monitor.js`. Let's place them in a new folder.

How to do it...

First, let's create our first TCP server in `server.js` as follows:

```
var net = require('net');
var fauxHttp = net.createServer(function(socket) {
    socket.write('Hello, this is TCP\n');
    socket.end();

    socket.on('data', function(data) {
        console.log(data.toString());
    });
}).listen(8080);
```

We can use the `nc` (netcat) command-line program to test this out in another terminal as follows:

```
echo "testing 1 2 3" | nc localhost 8080
```



If we're using Windows, we can download netcat from
<http://www.joncraton.org/blog/netcat-for-windows>.

The response should be `Hello, this is TCP\n` and the `server.js` console should output `testing 1 2 3`.

Remember, HTTP sits on top of TCP, so we can make an HTTP request of a TCP server. If we navigate our browser to `http://localhost:8080` and watch the console, we'll see all the headers from our browser's HTTP request appear in the console, with the browser displaying **Hello this is TCP**.

We've given the TCP server the name of `fauxHttp`. We're going to use it to record HTTP headers from browser clients (with some adjustments we could easily adapt our code to work with an actual HTTP server).

Still inside `server.js`, we're going to create another TCP server that opens a second port for `monitor.js` to communicate with our server. Before we do though, we'll make a new `EventEmitter` object as a bridge between our two `server.js` TCP servers:

```
var net = require('net'),
  stats = new (require('events').EventEmitter),
  filter = 'User-Agent';

var fauxHttp = net.createServer(function(socket) {
  socket.write('Hello, this is TCP\n');
  socket.end();

  socket.on('data', function(data) {
    stats.emit('stats', data.toString());
  });
});

fauxHttp.listen(8080);
```

We've replaced `console.log` in the `data` listener of `socket` with new `stats` `EventEmitter` which will emit a custom `stats` event upon receiving TCP data.

We also included a `filter` variable to be used in our second TCP interface in `server.js` as shown in the following code:

```
var monitorInterface = net.createServer(function(socket) {

  stats.on('stats', function(stats) {
    var header = stats.match(filter + ':') || stats.match('');
    header = header.input.substr(header.index).split('\r\n')[0];
    socket.write(header);
  });

  socket.write('Specify a filter [e.g. User-Agent]');
  socket.on('data', function(data) {
    filter = data.toString().replace('\n', '');
  });

  socket.write('Attempting to filter by: ' + filter);
});

monitorInterface.listen(8081);
```

Our monitor Interface server listens to our stats emitter to determine when the first TCP server has received information, sending this information (after it has been filtered) to a client connected on port 8081.

All we have to do now is create this client. Inside monitor.js we write the following code:

```
var net = require('net');
var client = net.connect(8081, 'localhost', function () {
  process.stdin.resume();
  process.stdin.pipe(client);
}).on('data', function (data) {
  console.log(data.toString());
}).on('end', function () {
  console.log('session ended');
});
```

When we open two terminals, running server.js in one and monitor.js in the other, and navigate to `http://localhost:8080` in our browser server.js transmits the User-Agent string from the HTTP headers of each request to monitor.js.

We can apply a different filter, such as `Accept`. By simply typing it into a running monitor.js process, any non-matching filters will default to returning the preliminary request line (GET /, POST /route/here, and so on).

To run across separate systems, we simply place server.js on a remote host and then update the second parameter of `net.connect` from localhost to the name of our server, for example:

```
var client = net.connect(8081, 'nodecookbook.com', function () {
```

How it works...

The HTTP layer works on top of TCP. So when our browser makes a request, the TCP server receives all the HTTP header information. `http.createServer` would handle these headers and other HTTP protocol interactions. However, `net.createServer` simply receives TCP data.

`socket.write` and `socket.end` are similar to `response.write` and `response.end` within an HTTP server callback function, but without reference to the requirements of the HTTP protocol. Nevertheless, the similarities are enough for our browser to be able to receive data from `socket.write`.

All our fauxHttp TCP server does is receive a request via port 8080, outputs Hello, this is TCP\n to the client, and reads any data from the client directly through to our stats event emitter.

Our `monitorInterface` TCP server listens on a separate port (8081), essentially giving us a sort of (completely insecure) admin interface. In the `monitorInterface` callback, we listen to the `stats` emitter which is triggered whenever a browser (or any TCP client) hits `localhost:8080`.

Inside the listener callback of `stats`, we retrieve the desired header, using the `filter` variable to search the HTTP headers with the `match` objects `index` and `input` properties, enabling us to extract the specified header. If there are no matches, we `match` an empty string, thereby returning a `match` object containing an `index` of 0 resulting in the extraction of the first line of the HTTP headers (the requested path and method).

The last part of the `monitorInterface` TCP server callback listens on the `socket` for a `data` event and sets the `filter` variable to whatever the client sends. This enables the `monitor.js` client to alter the `filter` by piping the `process.stdin` stream directly into the `TCP client`. Meaning we can type directly into `monitor.js`' running process and the `data` event from the `socket` of `monitorInterface` in `server.js`, and this will trigger `server.js` receiving whatever is typed in `monitor.js`' STDIN.

`monitor.js` avails of this functionality by piping the `process.stdin` stream directly into the `TCP client`. This means we can type directly into the running process and the `data` event from `socket` of `monitorInterface` in `server.js`, and this will trigger passing anything typed from STDIN of `monitor.js`.

There's more...

Let's look at some ways we can further harness the power of TCP.

Port forwarding

There can be various reasons to forward a port. As an example, if we wish to SSH into our server over a mobile connection, we may find that port 22 has been blocked. The same can apply with corporate firewalls (this could be because a blanket block is applied to all privileged ports except the most common such as 80 and 443).

We can use the `net` module to forward TCP traffic from one port to another, essentially circumventing a firewall. So naturally this should be used only for legitimate cases and with any necessary permission.

First, we'll require `net` and define ports to forward from and to:

```
var net = require('net');
var fromPort = process.argv[2] || 9000;
var toPort = process.argv[3] || 22;
```

So we can either define ports via command line or default to forwarding the arbitrary port 9000 to the SSH port.

Now we create a TCP server that receives connections via `fromPort`, creating a TCP client connection to `toPort`, passing all data between these connections as follows:

```
net.createServer(function (socket) {
  var client;
  socket.on('connect', function () {
    client = net.connect(toPort);
    client.on('data', function (data) {
      socket.write(data);
    });
  })
  .on('data', function (data) {
    client.write(data);
  })
  .on('end', function() {
    client.end();
  });
}).listen(fromPort, function () {
  console.log('Forwarding ' + this.address().port + ' to ' + toPort);
});
```

We use the `data` events to receive and push data between `client` (our bridge connection) and `socket` (the incoming connection).

If we now run our script on the remote server (with no arguments), we can log in to a secure shell from our local computer using port 9000 like so:

```
ssh -l username domain -p 9000
```

Using pcap to watch TCP traffic

With the third-party `pcap` module, we can also observe TCP packets as they travel in and out of our system. This can be useful for analysis and optimization of expected behaviors, performance, and integrity.

On the command line:

```
npm install pcap
```

For our code:

```
var pcap = require('pcap');
// may need to put wlan0, eth0, etc. as 1st arg. of createSession
var pcapSession = pcap.createSession("", "tcp");
var tcpTracker = new pcap.TCP_tracker();
```

```
tcpTracker.on('end', function (session) {
    console.log(session);
});

pcapSession.on('packet', function (packet) {
    tcpTracker.track_packet(pcap.decode.packet(packet));
});
```



If `pcap` fails to choose the correct device, there will be no output (or maybe unrelated output). In this case, we need to know which device to sniff. If we are connected wirelessly, it may well be `wlan0` or `wlan1` and if we are wired, in it could be `eth0/eth1`. We can find out by typing `ifconfig` (Linux, Mac OS X) or `ipconfig` (Windows) on the command line to see which device has an `inet address` matching the network part of our router's IP address (for example, `192.168.1.xxx`).

If we save this as `tcp_stats.js`, we can run it with the following:

```
sudo node tcp_stats.js
```

The `pcap` module interfaces with privileged ports and therefore must be run as root (for operating systems such as Linux and Mac OS X that enforce privileged ports).

If we navigate to any website and then refresh the page, the `tcpTracker.end` event of `pcap` is triggered in which we output the `session` object.

To initialize the `tcpTracker`, we create a `pcap` session and attach a listener for the `packet` event where we pass each decoded packet into `tcpTracker`.

Upon creating the `pcap` session we pass an empty string followed by `tcp` to the `createSession` method. The empty string causes `pcap` to automatically choose an interface (if this doesn't work we can specify the appropriate interface, for example, `eth0`, `wlan1` or `lo` if we want to analyze `localhost` TCP packets). The second parameter, `tcp`, instructs `pcap` to only listen for TCP packets.

See also

- ▶ *Creating an SMTP server* discussed in this chapter
- ▶ *Implementing a virtual hosting paradigm* discussed in this chapter

Creating an SMTP server

We don't have to rely on a third-party SMTP server, we can create our own!

In this recipe, we'll create our own internal SMTP server (just like the first SMTP servers) using the third-party `simpl.esmtp` module which is an underlying library of the `node-mailer` module from the first recipe of this chapter, *Sending email*. For information on converting an internal SMTP server to an externally exposed MX record server, see the *There's more...* section at the end of this recipe.

Getting ready

Let's create a file and call it `server.js`, then make a new folder called `mailboxes`, containing three subfolders: `bob`, `bob`, and `susie`. We'll also want to have our `mailout.js` file from the first recipe to hand.

How to do it...

First, we'll set up some initial variables:

```
var simpl.esmtp = require('simpl.esmtp');
var fs = require('fs');
var path = require('path');
var users = [{user: 'node', pass: 'cookbook'}],
  mailboxDir = './mailboxes/',
  catchall = fs.createWriteStream(mailboxDir + 'caught', {flags: 'a'});

```

Now, we initialize the SMTP server with authentication enabled:

```
var smtp = simpl.esmtp
  .createServer({requireAuthentication: true})
  .on('authorizeUser', function (envelope, user, pass, cb) {
    var authed;
    users.forEach(function (userObj) {
      if (userObj.user === user && userObj.pass === pass) {
        authed = true;
      }
    });
    cb(null, authed);
  });

```

Next, we'll react to some simple SMTP events to process incoming mail, beginning with the `startData` event:

```
smtplib.on('startData', function (envelope) {
  var rcpt, saveTo;
  envelope.mailboxes = [];
  envelope.to.forEach(function (to) {
    path.exists(mailboxDir + to.split('@')[0], function (exists) {
      rcpt = to.split('@')[0];
      if (exists) {
        envelope.mailboxes.unshift(rcpt);
        saveTo = mailboxDir + rcpt + '/' + envelope.from
          + ' - ' + envelope.date;
        envelope[rcpt] = fs.createWriteStream(saveTo, {flags: 'a'});
        return;
      }
      console.log(rcpt + ' has no mailbox, sending to catchall');
      envelope[rcpt] = catchall;
    });
  });
});
```

Then the `data` and `dataReady` events will be as follows:

```
smtplib.on('data', function (envelope, chunk) {
  envelope.mailboxes.forEach(function (rcpt) {
    envelope[rcpt].write(chunk);
  });
}).on('dataReady', function (envelope, cb) {
  envelope.mailboxes.forEach(function (rcpt) {
    envelope[rcpt].end();
  });

  cb(null, Date.now());
});
```

For terser code we chained these two events together with dot notation. Finally, we tell our SMTP server what port to listen on:

```
smtplib.listen(2525);
```

In production, it would be expedient to specify the port as 25 (or in more advanced cases 465 or 587).

Now let's test our server by converting our `mail.out.js` file from the *Sending email* recipe.

First, we alter our `createTransport` invocation to reflect the values of our custom SMTP server:

```
var transport = nodemailer.createTransport('SMTP', {  
  host: 'localhost',  
  secureConnection: false,  
  port: 2525,  
  auth: {  
    user: "node",  
    pass: "cookbook"  
  }  
});
```

Next, we modify the `maillist` array to reflect our mailboxes, as shown in the following code:

```
var maillist = [  
  'Bob <bob@nodecookbook.com>, Bib <bib@nodecookbook.com>',  
  'Mss Susie <susie@nodecookbook.com>',  
  'Mr Nobody <nobody@nodecookbook.com>',  
];
```

Bob and Bib are sent together. We also added an address which doesn't have a mailbox (`nobody@nodecookbook.com`) in order to test our catch all functionality.

Now, if we run `server.js` in one terminal and in another run `mail.out.js`, output from `mail.out.js` should be something like:

```
Sent to Mss Susie <susie@nodecookbook.com>  
Sent to Mr Nobody <nobody@nodecookbook.com>  
Sent to Bob <bob@nodecookbook.com>, Bib <bib@nodecookbook.com>
```

If we look in the `mailboxes/bob` directory, we'll see our email from `spamnot@ok.com`, the same for `susie` and `bib`.

`server.js` should have the following output:

```
nobody has no mailbox, sending to caught file
```

Therefore, upon analyzing the contents of `mailboxes/caught`, we'll see our email in there sent to Mr Nobody.

How it works...

SMTP is based upon a series of plain text communications between an SMTP client and server over a TCP connection. The `simpl.esntp` module carries out these communications for us, yielding a higher-level API for developer interactions.

When we call `simpl.esntp.createServer` with `requireAuthorization` set to `true`, our new server (simply called `sntp`) will emit an `authenticateUser` event and will not continue to process until we have invoked the fourth parameter, `cb` (the callback). `cb` takes two parameters. With the first we can specify a reason why access is denied via an `Error` object (we simply pass `null`). The second is a Boolean saying whether the user is authorized or not (if not, and the error parameter is null a generic access denied error is sent to the mail client).

We determine the second `cb` parameter by looping through our `users` array, finding out if the username and password are correct (in reality, we may wish to use a database for this part). If there is a match, our `auth` variable is set to `true` and passed to `cb`, otherwise it remains `false` and the client is rejected.

If the client is authorized, `sntp` will emit several events for each envelope (an envelope is an email package containing all the recipients for that email, body text, email headers, attachments, and so on).

In the `startData` event, we are provided with an `envelope` parameter where we use the `envelope.to` property to check whether our recipients have a mailbox. SMTP allows more than one recipient to be specified per email, so `envelope.to` is always an array, even if it contains only one recipient. Therefore, we use `for Each` to loop through `envelope.to` in order to check mailboxes for each recipient stipulated.

We find out the intended recipient mailbox by splitting the address by the @character, loading it into our `rcpt` variable. We perform no verification on the domain portion of the address, although `simpl.esntp` automatically verifies that the domain is genuine before emitting any of the events.

`rcpt` gets added to our `envelope.mailboxes` array, which we added to `envelope` before looping through `envelope.to`. We use `envelope.mailboxes` in the later `data` and `dataReady` events.

Still inside the `envelope.to` `for Each` loop, we add one final property to `envelope` named after the mailbox name (`rcpt`). If the mailbox exists, we create `writestream` to `saveTo` (a path with a filename determined from combining `envelope.from` with `envelope.date`). We now have an endpoint to each recipient's mailbox ready to receive data. If the mailbox doesn't exist for the recipient, we set `envelope[rcpt]` to `catchall.cat`. `catchall` is the global we set at the top of our file. It's a `writestream` with the `a` flag set so that the caught file accumulates orphaned emails. We create the `catchall.writestream` on initialization and then reuse the same `writestream` for all emails addressed to non-existent mailboxes. This saves us from creating a `writestream` for every badly addressed email received, thus saving resources.

The `data` event is triggered for each chunk of the email body received by the server, giving us `envelope` and `chunk`. We save each `chunk` to its applicable file using `envelope[rcpt].write`, determining the `rcpt` by looping through our custom `envelope.mailboxes` array.

The `dataReady` event signifies that all data has been received and the data is ready for processing. Since we've already stored it, we use this event to end the relevant `writestream` for each `rcpt` in our `mailboxes`. The `dataReady` event also requires a callback (`cb`). The first parameter can be an `Error` object, which allows for a final rejection of an email (for instance if the content of the email was analyzed and found to be spam). The second parameter expects a queue ID to send to the mail client, in our case we simply give `Date.now`.

There's more...

Let's take a look at how to convert our SMTP server into a public mail exchange handler.

Receiving emails from external SMTP servers

By removing authorization settings and remotely hosting our SMTP server, listening on port 25, we can allow other mail servers to communicate with our SMTP server so email can be transferred from one network to another (for example, from a Gmail account to our hosted SMTP server).

Let's save our file as `mx_smtp.js` and modify the following accordingly:

```
var simpleSmtp = require('simplesmtp');
var fs = require('fs');
var path = require('path');
var mailboxDir = './mailboxes/',
    catchall = fs.createWriteStream(mailboxDir + 'caught', {flags: 'a'});
var smtp = simpleSmtp.createServer();
```

We've discarded the `users` variable and changed the `smtp` variable so the object with the `requireAuthentification` property and the accompanying `authenticateUser` event are removed. In order for an external mail program to forward to our SMTP server it must be able to connect. Since other mail programs don't possess authentication details we have to open our server to allow them to do so.

The `startData` and `dataReady` events all remain the same. The final change is to the port:

```
smtp.listen(25);
```

In order for this to work, we must have a live server which we have root access to (for example, an amazon EC2 micro instance), and a domain where we can alter the mail exchange (MX) records.

So for instance, say we're hosting our SMTP server at `mysmtpserver.net` and we want to receive emails for `bob@nodecookbook.com`. We point the MX records of `nodecookbook.com` to `mysmtpserver.net` with a priority of 10.



For an example of how to change DNS records with a registrar, see `http://support.godaddy.com/help/article/680`. For more info on MX records, take a look at `http://en.wikipedia.org/wiki/MX_record`.

Once changes are made they can take a while to propagate (up to 48 hours, though often faster). We can use `dig mx` (Mac OS X and Linux) or `nslookup set q=MX` (Windows) to determine if the update to our MX records has occurred.

We must have Node installed on our remote host, ensuring that port 25 is exposed and not in use by any other programs. To check whether other programs are using port 25, log in with SSH and type `netstat -l`. If you see `*:smtp` in the **Active Internet Connections** (only servers) section, a program is already using the port and must be stopped (try `ps -ef` to look for any suspects).

On the live server, we create our `mailboxes` folder containing `bob bib` and `susie`, copy our `mx_smtp.js` file over, and install `simplesmtplib`.

```
npm install simplesmtplib
```

Now, if everything is properly set up, and our MX records are updated, we can execute our `mx_smtp.js` file on the live server. Then send a test email to `bob@nodecookbook.com` (or @whatever domain we possess which we have altered the MX records for), wait a few seconds, and then check the `mailboxes/bob` folder. The email should have appeared.

See also

- ▶ *Sending email discussed in this chapter*
- ▶ *Deploying to a server environment* discussed in Chapter 10, *Taking It Live*

Implementing a virtual hosting paradigm

If we wish to host multiple sites on one server, we can do so with virtual hosting. Virtual hosting is a way to uniquely handle multiple domain names according to their name. The technique is surprisingly simple: we just look at the incoming `Host` header and respond accordingly. In this task, we're going to implement simple name based virtual hosting for static sites.

Getting ready

We'll create a folder called `sites`, with `localhost-site` and `nodecookbook` as subdirectories. In `localhost-site/index.html` we'll write the following:

```
<b> This is localhost </b>
```

And in `nodecookbook/index.html` we'll add the following code:

```
<h1>Welcome to the Node Cookbook Site! </h1>
```

For local testing, we'll want to configure our system with some extra host names so we can point different domains to our server. To do this, we edit `/etc/hosts` on Linux and Max OS X, or `%SystemRoot%\system32\drivers\etc\hosts` for Windows systems.

At the top of the file it maps our local loopback IP 127.0.0.1 to `localhost`. Let's change this line to:

```
127.0.0.1 localhost nodecookbook
```

Finally, we want to create two new files: `mappings.js` and `server.js`. The `mappings.js` file will provide static file servers for each domain name, and `server.js` will provide the virtual hosting logic.

We'll be using the `node-static` module for serving our sites, our virtual host will only serve static websites. If we don't already have it, we can install it via npm as follows:

```
npm install node-static
```

How to do it...

Let's start with `mappings.js`:

```
var static = require('node-static');

function staticServe (dir) {
    return new (static.Server)('sites/' + dir)
}

exports.sites = {
    'nodecookbook' : staticServe('nodecookbook'),
    'localhost' : staticServe('localhost-site')
};
```

We've used the domains laid out in our system's `hosts` file. In a production scenario domains would be directed to us by DNS records.

Now for `server.js`:

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

var port = 8080,
    mappings = require('./mappings');

var server = http.createServer(function (req, res) {
    var domain = req.headers.host.replace(new RegExp(':' + port + '$'), '');
    site = mappings.sites[domain] ||
        mappings.sites[mappings_aliases[domain]];

    if (site) { site.serve(req, res); return; }
    res.writeHead(404);
    res.end('Not Found\n');

}).listen(port);
```

Now when we navigate to `http://localhost:8080` or `http://localhost:8080`, we get the content in `sites/localhost-site/index.html`. Whereas, if we go to `http://nodecookbook:8080`, we get the big Node Cookbook welcome message.

How it works...

Any time our server receives a request, we strip the port number (which wouldn't be necessary with port 80 servers) to determine the domain.

We then cross reference the domain with our `mappings.sites` object. If a site is found, we call its `serve` method, which is inherited from the `node-static` library. In `mappings.js`, each `exports.sites` property contains a `node-static Server` instance pointed at the relevant site directory. We've used our custom `staticServer` function as a wrapper to keep the code a little tidier.

To use the `static Server` instance, we call its `serve` method, passing through the `req` and `res` objects, as in `server.js`:

```
if (site) { site.serve(req, res); return; }
```

The `site` variable is an instance of `static.Server` pointing to the appropriate site folder for the given domain name.

If `server.js` fails to find a site in `mappings.js` for the requested domain, we simply pass a 404 error back to the client.

There's more...

What about going beyond static hosting into dynamic hosting, or what if we want to use an SSL/TLS certificate with our sites?

Virtual hosting Express apps

Express/Connect come with the `vhost` middleware which allows us to implement dynamic Express-based virtual hosting with ease.

First, we need to set up two Express apps. Let's remove the `nodecookbook` and `localhost-site` folders and regenerate them using the `express` binary as follows:

```
rm -fr nodecookbook && express nodecookbook
rm -fr localhost-site && express localhost-site
cd nodecookbook && npm -d install
cd ../../localhost-site && npm -d install
```

We'd also need to modify the last part of `app.js` in each site's file, wrapping the `app.listen` method in an `if module.parent` conditional:

```
if (!module.parent) {
  app.listen(3000);
  console.log("Express server listening on port %d in %s mode",
  app.address().port, app.settings.env);
}
```

In our `nodecookbook` app, let's add the following to `index.jade`:

```
h1 Welcome to the Node Cookbook site!
```

And in the `localhost-site` app, we add this code to `index.jade`:

```
b this is localhost
```

With our sites set up, we can modify `mappings.js` as follows:

```
function appServe(dir) {
  return require('../sites/' + dir + '/app.js')
}

exports.sites = {
  'nodecookbook' : appServe('nodecookbook'),
  'localhost' : appServe('localhost-site')
};
```

We've removed the `node-static` module since we're using Express instead. Our `staticServe` convenience function has been modified as `appServe`, which simply loads each Express app using `require` according to its mapping in `exports.servers`.

And we'll update `server.js` as follows:

```
var express = require('express'),
  mappings = require('./mappings'),
  app = express.createServer();

Object.keys(mappings.sites).forEach(function(domain) {
  app.use(express.vhost(domain, mappings.sites[domain]));
});

app.listen(8080);
```

We create a master app then loop through `mappings.sites`, passing each sub-app into `app.use` with `express.vhost`. The `vhost` middleware takes two arguments. The first is the domain. We get each domain from the `mappings.sites` keys. The second is an Express app. We retrieve each Express app from the values in `mappings.sites`.

We simply request the domain and the `vhost` middleware lines up the relevant domain with the relevant app to deliver the correct site.

Server Name Indication

Prior to Server Name Indication (SNI), name-based virtual hosting for sites served over SSL/TLS was a complex administration issue (requiring every hostname to be stored in a multi-domain certificate).

This is because an encrypted connection is established based upon a certificate that specifies a domain name before any HTTP headers are received by the server. Therefore, the server cannot provide a certificate that is specific to one domain. As a result, the browser would vividly warn the user that the connection may be unsafe since the domain name listed on the certificate didn't match the domain being accessed. In order to avoid this scenario, a virtual host would have to buy a certificate containing every domain hosted, then reapply for a new certificate every time a new domain was added or removed.

SNI forwards the requested domain to the server at the beginning of the SSL/TLS handshake, allowing our server to choose the appropriate certificate for a domain, and preventing the browser from telling our users that they may be under attack.

`https.Server` (inheriting from `tls.Server`) has the `addContext` method, which allows us to specify hostname and certificate credentials for multiple individual domains.

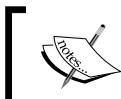
Let's enable TLS-compatible virtual hosting by making a few changes. First, in `mappings.js`, we'll add another convenience function called `secureShare`:

```
function secureShare(domain) {
  var site = {
    content: staticServe(domain),
    cert: fs.readFileSync('sites/' + domain + '/certs/cert.pem'),
    key: fs.readFileSync('sites/' + domain + '/certs/key.pem')
  };
  return site;
}
```

Next, we'll alter the way we load the sites, calling `secureShare` instead of `staticServe`:

```
exports.sites = {
  'nodecookbook.com' : secureShare('nodecookbook.com'),
  'davidmarkelements.com' : secureShare('davidmarkelements.com')
};
```

For this example to work in a production scenario, we'll have to replace the sample domains with the ones that we control and obtain genuine certificates signed by a trusted Certificate Authority.



We can test locally by following the instructions in the supporting code files for this chapter (under `secure_virtual_hosting/howto`).

Let's change our `sites` folder structure to conform to alterations made in `mappings.js` by renaming `nodecookbook` to `nodecookbook.com` and `localhost-site` to `davidmarkelements.com`, changing the latter's `index.html` file to the following:

```
<b>This is DavidMarkElements.com virtually AND secure</b>
```

Each `site` folder also needs a `certs` folder containing our `cert.pem` and `key.pem` files. These files must be certificates purchased specifically for the domain.

In `server.js`, we change the top of our script to the following:

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

The `filesystem(fs)` module is required to load our credentials. As we've replaced `http` with `https`, we will alter our `createServer` call as follows:

```
var server = https.createServer({}, function (req, res) {
```

Simply adding 's' to `http` does the trick. On this occasion, there's no SSL/TLS credentials since we'll be loading these on a per domain basis with the `addContext` method. Therefore, we just pass an empty object.

In `mappings.js`, our `secureShare` function returns an object containing three properties, `content`, `key`, and `cert`, where `content` holds the static server. So in `server.js`, we update this line:

```
if (site) { site.serve(req, res); return; }
```

To:

```
if (site) { site.content.serve(req, res); return; }
```

As we're hosting on a live server, we expose it to incoming web connections by binding to `0.0.0.0`:

```
).listen(port, '0.0.0.0');
```

We could also change the `port` variable to 443 to serve directly over the HTTPS port (we must run the server as root to do this, which has security implications in a live environment, see *Chapter 10, Taking It Live*, for how to do this safely).

And finally, we add the following to the bottom of `server.js`:

```
Object.keys(mappings.sites).forEach(function (hostname) {
  server.addContext(hostname, {key: mappings.sites[hostname].key,
    cert: mappings.sites[hostname].cert});
});
```

This loads the `key` and `cert` properties for each domain based on the settings laid out in `mappings.js`.

Provided we have trusted CA-certified credentials for each specified domain, and we're using a modern browser, we can navigate to each site using HTTPS without a receiving a warning.

The Caveat

There is a catch: Server Name Indication only works in modern browsers. Although modern browsers in this context exclude Internet Explorer 7/8 and Safari when run on Windows XP, as well as Android Gingerbread (versions 2.x) and Blackberry browsers. If we provided a default certificate via the options object of `htps.createServer`, the user will still be able to view the site on older browsers, but they will receive the same warnings as if we weren't using SNI (the older browsers don't indicate the hostname in SSL/TLS negotiations so our SNI handling never occurs). Depending on the intended market, we may have to use alternative methods until these older browsers are used in sufficiently low numbers in relation to our purposes.

See also

- ▶ *Serving static files* discussed in *Chapter 1, Making a Web Server*
- ▶ *Dynamic routing* discussed in *Chapter 6, Accelerating Development with Express*
- ▶ *Setting up an HTTPS web server* discussed in *Chapter 7, Implementing Security, Encryption, and Authentication*
- ▶ *Deploying to a server environment* discussed in *Chapter 10, Taking It Live*

9

Writing Your Own Node Modules

In this chapter we will cover:

- ▶ Creating a test-driven module API
- ▶ Writing a functional module mock-up
- ▶ Refactoring from functional to prototypical
- ▶ Extending a module's API
- ▶ Deploying a module to npm

Introduction

A thriving module ecosystem has been one of the core goals of Node since its inception. The framework leans heavily toward modularization. Even the core functionality (such as HTTP) is made available through the module system.

It's almost as easy to create our own modules as it is to use core and third-party modules. All we need to know is how the module system works and a few best practices.

A great module is one that performs a specific function to a high standard and great code is the result of multiple development cycles. In this chapter, we're going to develop a module from scratch, beginning with defining its Application Programming Interface (API), to creating our module over a series of development cycle iterations. We'll finally deploy it to npm for the benefit of all.

Creating a test-driven module API

We're going to create our module by loosely following the test-driven development (TDD) model (See http://en.wikipedia.org/wiki/Test-driven_development for more info). JavaScript is asynchronous so code can be executed in multiple time streams at once. This can sometimes make for a challenging mental puzzle.

A test suite is a particularly powerful tool when it comes to JavaScript development. It provides a quality assurance process and inspires confidence in a module's users base when tests are passed.

What's more, we can define our tests up front as a way to map out the intended API before we even begin development.

In this recipe, we'll be doing just that by creating a test suite for a module that extracts stats information from MP3 files.

Getting ready

Let's create a new folder called `mp3dat`, with a file inside called `index.js`. Then two subfolders: `lib` and `test`, both containing `index.js`.

We'll also need MP3 files to test on. For simplicity, our module will only support MPEG-1 Layer 3 files with error protection turned off. Other types of MP3 files include MPEG-2 and MPEG-2.5. MPEG-1 (no error protection) would be the most common type, but our module could easily be extended later. We can retrieve an MPEG-1 Layer 3 file from <http://www.paulsladen.org/pronunciation/torvalds-says-linux.mp3>. Let's place this file in our new `mp3dat/test` folder and call it `test.mp3`.

The focus of this chapter is to create a fully functioning module, prior knowledge of MP3 file structures is not required. Details regarding MP3s in this chapter can safely be scanned over while the information pertaining to module creation is of key importance. However, we can learn more about MP3 files and their structure from <http://en.wikipedia.org/wiki/MP3>.

How to do it...

Let's open `test/index.js` and set up some variables as shown in the following code:

```
var assert = require('assert');
var mp3dat = require('../index.js');
var testFile = 'test/test.mp3';
```

`assert` is a core Node module specifically for building test suites. The general idea is we `assert` that something should be true (or false), and if the assertion is correct the test passes. The `mp3dat` variable requires our primary (currently blank) `index.js` file that will in turn load the `lib/index.js` file, which holds the actual module code.

The `testFile` variable points to our `test.mp3` file from the perspective of the root of our module (the `mp3dat` folder). This is because we run our tests from the root of the module directory.

Now we'll decide our API and write the corresponding tests. Let's model our module after the `fs.stat` method. We'll retrieve data about the MP3 file using an `mp3dat.stat` method, which will take two arguments: a file path and a callback function to be invoked once the stats have been gathered.

The `mp3dat.stat` callback will take two arguments. The first will be the error object which should be set to `null` if there is no error, and the second will contain our `stats` object.

The `stats` object will contain `duration`, `bitrate`, `filesize`, `timestamp`, and `timesig` properties. The `duration` property will in turn contain an object holding `hours`, `minutes`, `seconds`, and `milliseconds` keys.

For example, our `test.mp3` file should return something like the following:

```
{ duration: { hours: 0, minutes: 0, seconds: 5, milliseconds: 186 },
  bitrate: 128000,
  filesize: 82969,
  timestamp: 5186,
  timesig: '00:00:05' }
```

Now that we've conceptualized our API, we can map it out to assertion tests as a means of enforcing that API throughout the modules development.

Let's start with `mp3dat` and `mp3dat.stat`.

```
assert(mp3dat, 'mp3dat failed to load');

assert(mp3dat.stat, 'there should be a stat method');

assert(mp3dat.stat instanceof Function, 'stat should be a Function');
```

To test the `mp3dat.stat` function we actually have to call it, then we perform further tests within its callback.

```
mp3dat.stat(testFile, function (err, stats) {

  assert.ifError(err);

  // expected properties
  assert(stats.duration, 'should be a truthy duration property');
  assert(stats.bitrate, 'should be a truthy bitrate property');
  assert(stats.filesize, 'should be a truthy filesize property');
  assert(stats.timestamp, 'should be a truthy timestamp property');
  assert(stats.timesig, 'should be a truthy timesig property');
```

Now that we've established expected `stats` properties, we can go further and specify what these properties should look like, still within the callback we write the following code:

```
//expected types
assert.equal(typeof stats.duration, 'object', 'duration should be an
object type');
assert(stats.duration instanceof Object, 'durations should be an
instance of Object');
assert(!isNaN(stats.bitrate), 'bitrate should be a number');
assert(!isNaN(stats.filesize), 'filesize should be a number');
assert(!isNaN(stats.timestamp), 'timestamp should be a number');

assert(stats.timesig.match(/^\d+:\d+:\d+$/), 'timesig should be in
HH:MM SS format');

//expected duration properties
assert.notStrictEqual(stats.duration.hours, undefined, 'should be a
duration.hours property');
assert.notStrictEqual(stats.duration.minutes, undefined, 'should be a
duration.minutes property');
assert.notStrictEqual(stats.duration.seconds, undefined, 'should be a
duration.seconds property');
assert.notStrictEqual(stats.duration.milliseconds, undefined,
'should be a duration.milliseconds property');

//expected duration types
assert(!isNaN(stats.duration.hours), 'duration.hours should be a
number');
assert(!isNaN(stats.duration.minutes), 'duration.minutes should be a
number');
assert(!isNaN(stats.duration.seconds), 'duration.seconds should be a
number');
assert(!isNaN(stats.duration.milliseconds), 'duration.milliseconds
should be a number');

//expected duration properties constraints
assert(stats.duration.minutes < 60, 'duration.minutes should be no
greater than 59');
assert(stats.duration.seconds < 60, 'duration.seconds should be no
greater than 59');
assert(stats.duration.milliseconds < 1000, 'duration.seconds should
be no greater than 999');

console.log('All tests passed'); //if we've gotten this far we are
done.
});
```

Now let's run our test. From the `mp3dat` folder we say:

```
node test
```

This should return text containing the following:

```
AssertionError: there should be a stat method
```

Which is exactly right, we haven't written the `stat` method yet.

How it works...

When the tests are run, the `assert` module will throw `AssertionError` to let the developer know that their code is not currently lining up with their predefined assertions regarding the desired API.

In our unit test file (`test/index.js`), we mainly used the simple `assert` function (an alias for `assert.ok`). `assert` requires that the first argument passed to it be truthy. Otherwise it throws `AssertionError` where the second provided argument is given for the error message (the opposite of `assert.ok` is `assert.fail`, which expects a falsey value).

Our test fails at:

```
assert(mp3dat.stat, 'there should be a stat method');
```

This is because `mp3dat.stat` is undefined (a falsey value).

The first argument of `assert` can be an expression. For instance, we use `stats.duration.minutes < 60` to set a constraint for the `duration.minutes` property, and use the `match` method on `timesig` to verify a correct time pattern of HH:MM:SS.

We also use `assert.equal` and `assert.notStrictEqual`. `assert.equal` is a test that applies equality with type coercion (e.g. equivalent to `==`) and `assert.strictEqual` requires that values and types match – `assert.notEqual` and `assert.notStrictEqual` are the corresponding antipathies.

We use `assert.notStrictEqual` to ensure the existence of the `duration` object's sub-properties (`hours`, `minutes`, and so on).

There's more...

There are many testing frameworks providing extra descriptive syntax, enhanced functionality, asynchronous testing capabilities, and more. Let's sample one.

Unit tests with `should.js`

The third-party `should` module sits nicely on top of the core `assert` module, adding some syntactic sugar to both simplify and increase the descriptive powers of our tests. Let's install it.

```
npm install should
```

Now we can rewrite our tests with `should` as shown in the following code:

```
var should = require('should');
var mp3dat = require('../index.js');
var testFile = 'test/test.mp3';

should.exist(mp3dat);
mp3dat.should.have.property('stat');
mp3dat.stat.should.be.an.instanceof(Function);

mp3dat.stat(testFile, function (err, stats) {
  should.ifError(err);

  // expected properties
  stats.should.have.property('duration');
  stats.should.have.property('bitrate');
  stats.should.have.property('filesize');
  stats.should.have.property('timestamp');
  stats.should.have.property('timesig');

  // expected types
  stats.duration.should.be.an.instanceof(Object);
  stats.bitrate.should.be.a('number');
  stats.filesize.should.be.a('number');
  stats.timestamp.should.be.a('number');

  stats.timesig.should.match(/^\d+:\d+:\d+$/);

  // expected duration properties
  stats.duration.should.have.keys('hours', 'minutes', 'seconds',
    'milliseconds');

  // expected duration types and constraints
  stats.duration.hours.should.be.a('number');
  stats.duration.minutes.should.be.a('number').and.be.below(60);
  stats.duration.seconds.should.be.a('number').and.be.below(60);
  stats.duration.milliseconds.should.be.a('number').and.
    be.below(1000);

  console.log('All tests passed');
});


```

`should` allows us to write more concise and descriptive tests. Its syntax is natural and self-explanatory. We can read up on various `should` methods at its Github page:
<https://www.github.com/visionmedia/should.js>.

See also

- ▶ *Writing a functional module mock-up* discussed in this chapter
- ▶ *Extending the module's API* discussed in this chapter
- ▶ *Deploying a module to npm* discussed in this chapter

Writing a functional module mock-up

Now that we have our tests written (see previous recipe), we are ready to create our module (incidentally, from here on we'll be using the `should` version of our unit tests as opposed to `assert`).

In this recipe, we'll write our module in simple functional style to demonstrate proof of concept. In the next recipe, we'll refactor our code into a more common modular format centered on reusability and extendibility.

Getting ready

Let's open our main `index.js` and link it to the `lib` directory via `module.exports`.

```
module.exports = require('./lib');
```

This allows us to place the meat of our module code neatly inside the `lib` directory.

How to do it...

We'll open up `lib/index.js` and begin by requiring the `fs` module, which will be used to read an MP3 file, and setting up a `bitrates` map that cross references hex-represented values to bitrate values as defined by the MPEG-1 specification.

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

// half-byte (4bit) hex values to interpreted bitrates (bps)
// only MPEG-1 bitrates supported
var bitrates = { 1: 32000, 2: 40000, 3: 48000, 4: 56000, 5: 64000,
  6: 80000, 7: 96000, 8: 112000, 9: 128000, A: 160000, B: 192000,
  C: 224000, D: 256000, E: 320000 };
```

Now we'll define two functions, `findBitRate` to locate and translate the bitrate half-byte and `buildStats` to crunch all gathered information into our previously determined final `stats` object.

```
function buildStats(bitrate, size, cb) {
  var magnitudes = [ 'hours', 'minutes', 'seconds', 'milliseconds' ],
    duration = {}, stats,
    hours = (size / (bitrate / 8) / 3600);

  (function timeProcessor(time, counter) {
    var timeArray = [], factor = (counter < 3) ? 60 : 1000;
    if (counter) {
      timeArray = (factor * +(0.' + time)).toString().split('.');
    }

    if (counter < magnitudes.length - 1) {
      duration[magnitudes[counter]] = timeArray[0] || Math.floor(time);
      duration[magnitudes[counter]] =
        +duration[magnitudes[counter]];
      counter += 1;
      timeProcessor(timeArray[1] || time.toString().split('.')[1],
        counter);
      return;
    }
    // round off the final magnitude
    duration[magnitudes[counter]] = Math.round(timeArray.join('.'));
  })(hours, 0));

  stats = {
    duration: duration,
    bitrate: bitrate,
    filesize: size,
    timestamp: Math.round(hours * 3600000),
    timesig: ''
  };

  function pad(n){return n < 10 ? '0'+n : n}
  magnitudes.forEach(function (mag, i) {
    if (i < 3) {
      stats.timesig += pad(duration[mag]) + ((i < 2) ? ':' : '');
    }
  });

  cb(null, stats);
}
```

`bui l dSt at s` takes `bit rate`, `size`, and `cb` arguments. It uses `bit rate` and `size` to calculate the amount of seconds in the track and then uses this information to generate the `st at s` object, which it passes through the `cb` function.

To get `bit rate` into `bui l dSt at s`, let's write the `findBitRate` function as shown in the following code:

```
function findBitRate(f, cb) {
  fs.createReadStream(f)
    .on('data', function (data) {
      var i;
      for (i = 0; i < data.length; i += 2) {
        if (data.readUInt16LE(i) === 64511) {
          this.destroy();
          cb(null, bitrates[data.toString('hex', i + 2, i + 3)[0]]);
          break;
        };
      }
    }).on('end', function () {
      cb(new Error('could not find bitrate, is this definitely an MPEG-1
MP3?'));
    });
}
}
```

Finally, we expose a `st at` method, which utilizes our functions to produce the `st at s` object:

```
exports.st at = function (f, cb) {
  fs.st at(f, function (err, fstats) {
    findBitRate(f, function (err, bitrate) {
      if (err) { cb(err); return; }
      buildStats(bitrate, fstats.size, cb);
    });
  });
}
```

Now let's run our (should) tests from the previous recipe:

```
node test
```

It should output the following:

```
All tests passed
```

How it works...

The `exports` object is a central part of the Node platform. It's the other half of `require`. When we require a module, any properties added to `exports` are exposed through `require`. So when we do:

```
var mp3dat = require('mp3dat');
```

We can access `exports.stat` through `mp3dat.stat`, or even through `require('mp3dat').stat` (assuming we have `mp3dat` installed as a module, see *Deploying a module to npm*).

If we want to expose one function for the entire module, we use `module.exports`, as in our top `index.js` file which we set up in the *Getting ready* section of this recipe.

`Our stat method first calls fs.stat with the user-supplied filename (f). We use the supplied stats object to retrieve the size of our file, which we pass to buildStats. That is, after we have called findBitRate to retrieve the MP3's bitrate, which we also pass to buildStats.`

The `buildStats` callback is passed straight up through to our `stat` method's callback, the execution of the user callback originates within `buildStats`.

`findBitRate creates a readStream of the user-supplied file (f) and loops through each emitted data chunk two bytes at a time, thus halving the search time. We can do this because we're looking for a two-byte sync word, which will always be at a position divisible by two. In hex the sync word is FFFB, as a 16-byte little-endian unsigned integer it's 64511 (this is true only for MPEG-1 MP3 files without error protection).`

The next four bits (half-byte) following the MP3 sync word contain the bitrate value. So we pass this through the `Buffer.toString` method, requiring hex output which we match against our `bitrates` object map. In the case of our `test.mp3` file, the half-byte hex value is 9 representing a bitrate of 128000 bits per second.

Once we find our bitrate, we execute the callback and invoke `this.destroy`, which abruptly terminates our `readStream`, preventing the `end` event from ever being triggered. The `end` event will only occur when a bitrate has not been discovered, in which case we send an error back through the callback.

`buildStats receives bitrate and divides it by 8 giving us the bytes per second (8 bits to a byte). Dividing the total bytes of the MP3 by the bytes per second renders the amount of seconds. We then further divide it by 3,600 to get the hours variable, which is then passed into the embedded timeProcessor function. timeProcessor simply recurses through the magnitudes array (hours, minutes, seconds, milliseconds) until seconds have been accurately converted and apportioned to each magnitude, which gives us our duration object. Again, we use the calculated duration (in whatever form) to construct our tempo and timesig properties.`

There's more...

Examples of how to use a module can be a great resource for end users. Let's write an example for our new module.

Writing a module use case example

We'll create an `examples` folder within the `mp3dat` folder, and create a file called `basic.js` (for a basic usage example) writing the following into it:

```
var mp3dat = require('../index.js');

mp3dat.stat('../test/test.mp3', function (err, stats) {
  console.log(stats);
});
```

This should cause the console to output the following:

```
{ duration: { hours: 0, minutes: 0, seconds: 5, milliseconds: 186 },
  bitrate: 128000,
  filesize: 82969,
  timestamp: 5186,
  timesig: '00:00:05' }
```

See also

- ▶ *Creating a test-driven module API* discussed in this chapter
- ▶ *Refactoring from functional to prototypical* discussed in this chapter
- ▶ *Deploying a module to npm* discussed in this chapter

Refactoring from functional to prototypical

The functional mock-up created in the previous recipe can be useful for gaining mental traction with a concept (that is, getting our head around it), and may be perfectly adequate for small, simple modules with narrow scope.

However, the prototype pattern (among others) is commonly used by module creators, often used in Node's core modules and is fundamental to native JavaScript methods and objects.

Prototypical inheritance is marginally more memory efficient. Methods sitting on a prototype are not instantiated until called, and they're reused instead of recreated on each invocation.

On the other hand, it can be slightly slower than our previous recipe's procedural style because the JavaScript engine has the added overhead of traversing prototype chains. Nevertheless, it's (arguably) more appropriate to think of and implement modules as entities in their own right, which a user can create instances of (for example, a prototype-oriented approach). For one, it makes them easier to programmatically extend through cloning and prototype modifications. This leads to great flexibility being afforded to the end user while the core integrity of the module's code stays intact.

In this recipe, we'll rewrite our code from the previous task according to the prototype pattern.

Getting ready

Let's start editing `index.js` in `mp3dat/lib`.

How to do it...

To begin, we'll need to create a constructor function (a function called using `new`), which we'll name `Mp3dat`:

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

function Mp3dat(f, size) {
  if (!(this instanceof Mp3dat)) {
    return new Mp3dat(f, size);
  }
  this.stats = {duration: {}};
}
```

We've also required the `fs` module as in the previous task.

Let's add some objects and methods to our constructor's prototype:

```
Mp3dat.prototype._bitrates = { 1 : 32000, 2 : 40000, 3 : 48000, 4 : 56000, 5 : 64000, 6 : 80000, 7 : 96000, 8 : 112000, 9 : 128000, A : 160000, B : 192000, C : 224000, D : 256000, E : 320000 };

Mp3dat.prototype._magnitudes = [ 'hours', 'minutes', 'seconds',
'milliseconds' ];

Mp3dat.prototype._pad = function (n) { return n < 10 ? '0' + n : n; }

Mp3dat.prototype._timesig = function () {
  var ts = '', self = this;;
  self._magnitudes.forEach(function (mag, i) {
    if (i < 3) {
```

```

        ts += self._pad(self.stats.duration[mag]) + ((i < 2) ? ':' : '');
    }
});
return ts;
}

```

Three of our new `Mp3dat` properties (`_magnitudes`, `_pad`, and `_timesig`) were contained in the `buildStats` function in some form. We've prefixed their names with the underscore (`_`) to signify that they are private. This is merely a convention, JavaScript doesn't actually privatize them.

Now we'll reincarnate the previous recipe's `findBitRate` function as follows:

```

Mp3dat.prototype._findBitRate = function(cb) {
    var self = this;
    fs.createReadStream(self.f)
        .on('data', function(data) {
            var i = 0;
            for (i; i < data.length; i += 2) {
                if (data.readUInt16LE(i) === 64511) {
                    self.bitrate = self._bitrates[data.toString('hex', i + 2,
                        i + 3)[0]];
                    this.destroy();
                    cb(null);
                    break;
                };
            }
        }).on('end', function() {
            cb(new Error('could not find bitrate, is this definitely an MPEG-1
MP3?'));
        });
    }
}

```

The only differences here are that we load the filename from the object (`self.f`) instead of via the first parameter, and we load `bitrate` onto the object instead of sending it through the second parameter of `cb`.

Now to convert `buildStats` into the prototype pattern, we write the following code:

```

Mp3dat.prototype._buildStats = function(cb) {
    var self = this,
        hours = (self.size / (self.bitrate / 8) / 3600);

    self._timeProcessor(hours, function(duration) {
        self.stats = {
            duration: duration,
            bitrate: self.bitrate,

```

```
    filesize: self.size,
    timestamp: Math.round(hours * 3600000),
    timesig: self._timesig(duration, self.magnitudes)
  };
  cb(null, self.stats);
}

});

}
```

Our `_buildStats` prototype method is significantly smaller than its `buildStats` cousin from the previous task. Not only have we pulled its internal `magnitudes` array, pad utility function, and time signature functionality (wrapping it into its own `_timesig` method), we've also outsourced the internal recursive `timeProcessor` function to a prototype method equivalent.

```
Mp3dat.prototype._timeProcessor = function (time, counter, cb) {
  var self = this, timeArray = [], factor = (counter < 3) ? 60 : 1000,
    magnitudes = self._magnitudes, duration = self.stats.duration;

  if (counter instanceof Function) {
    cb = counter;
    counter = 0;
  }

  if (counter) {
    timeArray = (factor * +'0.' + time).toString().split('.');
  }
  if (counter < magnitudes.length - 1) {
    duration[magnitudes[counter]] = timeArray[0] || Math.floor(time);
    duration[magnitudes[counter]] = +duration[magnitudes[counter]];
    counter += 1;
    self._timeProcessor.call(self, timeArray[1] || time.toString().
      split('.')[1], counter, cb);
    return;
  }
  // round off the final magnitude (milliseconds)
  duration[magnitudes[counter]] = Math.round(timeArray.join('.'));
  cb(duration);
}
```

Finally, we write the `stat` method (with no underscore prefix since it's intended for public use), and export the `Mp3dat` object.

```
Mp3dat.prototype.stat = function (f, cb) {
  var self = this;
  fs.stat(f, function (err, fstats) {
```

```
self.size = fstats.size;
self.f = f;
self._findBitRate(function (err, bitrate) {
  if (err) { cb(err); return; }
  self._buildStats(cb);
});
});
}
}

module.exports = Mp3dat();
```

We can ensure all is present and correct by running the tests we built in the first recipe. On the command line from the `mp3dat` folder we say:

```
node test
```

Which should output:

```
All tests passed
```

How it works...

In the previous recipe, we had an `exports.stat` function which called the `findBitRate` and `buildStats` functions to get the `stats` object. In our refactored module, we add the `stat` method onto the prototype and export the entire `Mp3dat` constructor function via `module.exports`.

We don't have to pass `Mp3dat` to `module.exports` using `new`. Our function generates the new instance when invoked directly, with the following code:

```
if (!(this instanceof Mp3dat)) {
  return new Mp3dat();
}
```

This is really a failsafe strategy. It's more efficient (though marginally) to initialize the constructor with `new`.

The `stat` method in our refactored code differs from the `exports.stat` function in the prior task. Instead of passing the filename and size of the specified MP3 as parameters to `findBitRate` and `buildStats` respectively, it assigns them to the parent object via `this` (which is assigned to `self` to avoid new callbacks scopes reassignment of `this`).

It then invokes the `_findBitRate` and `_buildStats` methods to ultimately generate the `stats` object and pass it back to the users callback.

After running `mp3dat.stats` on our `test.mp3` file, our refactored `mp3dat` module object will hold the following:

```
{ stats:
  { duration: { hours: 0, minutes: 0, seconds: 5, milliseconds: 186 },
    bitrate: 128000,
    filesize: 82969,
    timestamp: 5186,
    timesig: '00:00:05' },
  size: 82969,
  f: 'test/test.mp3',
  bitrate: 128000 }
```

In the former recipe however, the returned object would simply be as follows:

```
{ stat: [Function] }
```

The functional style reveals the API. Our refactored code allows the user to interact with the information in multiple ways (through the `stats` and `mp3dat` objects). We can also extend our module and populate `mp3dat` with other properties later on, outside of the `stats` object.

There's more...

We can structure our module to make it even easier to use.

Adding the stat function to the initialized mp3dat object

If we want to expose our `stat` function directly to the `mp3dat` object, thus allowing us to view the API directly (for example, with `console.log`), we can add it by removing `Mp3dat.prototype.stat` and altering `Mp3dat` as follows:

```
function Mp3dat() {
  var self = this;
  if (!(this instanceof Mp3dat)) {
    return new Mp3dat();
  }
  self.stat = function (f, cb) {
    fs.stat(f, function (err, fstats) {
      self.size = fstats.size;
      self.f = f;
      self._findBitRate(function (err, bitrate) {
        if (err) { cb(err); return; }
        self._buildStats(cb);
      });
    });
  }
  self.stats = {duration: {}};
}
```

Then our final object becomes:

```
{ stat: [Function],  
  stats:  
    { duration: { hours: 0, minutes: 0, seconds: 5, milliseconds: 186  
    },  
      bitrate: 128000,  
      filesize: 82969,  
      timestamp: 5186,  
      timesig: '00:00:05' },  
    size: 82969,  
    f: 'test/test.mp3',  
    bitrate: 128000 }
```

Alternatively, if we're not concerned about pushing the `stats` object and other `Mp3dat` properties through to the module user, we can leave everything as it is, except change the following code:

```
module.exports = Mp3dat()
```

To:

```
exports.stat = function (f, cb) {  
  var m = Mp3dat();  
  return Mp3dat.prototype.stat.call(m, f, cb);  
}
```

This uses the `call` method to apply the `Mp3dat` scope to the `stat` method (allowing us to piggyback off of the `stat` method) and will return an object with the following:

```
{ stat: [Function] }
```

Just as in the first write of our module, except we still have the prototype pattern in place. This second approach is ever so slightly more efficient.

Allowing for multiple instances

Our module is a singleton since returns the already initialized `Mp3dat` object. This means no matter how many times we require it and assign it to variables, a module user will always be referring to the same object, even if `Mp3dat` is required in different submodules loaded by a parent script.

This means bad things will happen if we try to run two `mp3dat.stat` methods at the same time. In a situation where our module is required multiple times, two variables holding the same object could end up overwriting each other's properties, resulting in unpredictable (and frustrating) code. The most likely upshot is that `readStreams` will clash.

One way to overcome this is to alter the following:

```
module.exports = Mp3dat()
```

To:

```
module.exports = Mp3dat
```

And then load two instances with the following code:

```
var Mp3dat = require('../index.js'),
    mp3dat = Mp3dat(),
    mp3dat2 = Mp3dat();
```

If we wanted to provide both singletons and multiple instances, we could add a `spawnInstance` method to our constructor's prototype:

```
Mp3dat.prototype.spawnInstance = function () {
    return Mp3dat();
}

module.exports = Mp3dat();
```

Which then allows us to do something as follows:

```
var mp3dat = require('../index.js'),
    mp3dat2 = mp3dat.spawnInstance();
```

Both `mp3dat` and `mp3dat2` would be separate `Mp3dat` instances, whereas in the following case:

```
var mp3dat = require('../index.js'),
    mp3dat2 = require('../index.js');
```

Both would be the same instance.

See also

- ▶ *Writing a functional module mock-up* discussed in this chapter
- ▶ *Extending the module's API* discussed in this chapter
- ▶ *Deploying a module to npm* discussed in this chapter

Extending the module's API

There are many ways we can extend our module, for example, we could make it support more MP3 types, but this is merely leg work. It just takes finding out the different sync words and bitrates for different types of MP3, and then adding these to the relevant places.

For a more interesting venture, we could extend the API, creating more options for our module users.

Since we use a stream to read our MP3 file, we could allow the user to pass in either a filename or a stream of MP3 data, offering both ease (with a simple filename) and flexibility (with streams). This way we could start a download stream, STDIN stream, or in fact any stream of MP3 data.

Getting ready

We'll pick up our module from where we left it at the end of *Allowing for multiple instances* in the *There's more...* section of the previous recipe.

How to do it...

First, we'll add some more tests for our new API. In `tests/index.js`, we'll pull out the callback function from the `mp3dat.stat` call into the global scope, and we'll call it `cb`:

```
function cb (err, stats) {
    should.ifError(err);

    // expected properties
    stats.should.have.property('duration');

    // ...all the other unit tests here

    console.log('passed');

};
```

Now we'll call `stat` along with a method which we're going to write and name: `statStream`

```
mp3dat.statStream({stream: fs.createReadStream(testFile),
    size: fs.statSync(testFile).size}, cb);

mp3dat2.stat(testFile, cb);
```

Notice we're using two `Mp3dat` instances (`mp3dat` and `mp3dat2`). So we can run `stat` and `statStream` tests side by side. Since we're creating a `readStream`, we require `fs` at the top of our `[tests/index.js]` file.

```
var should = require('should');
var fs = require('fs');
var mp3dat = require('../index.js'),
    mp3dat2 = mp3dat.spawnInstance();
```

We'll also put a few top-level should tests in for the stat Stream method as follows:

```
should.exist(mp3dat);
mp3dat.should.have.property('stat');
mp3dat.stat.should.be.an.instanceof(Function);
mp3dat.should.have.property('statStream');
mp3dat.statStream.should.be.an.instanceof(Function);
```

Now to live up to our tests expectations.

Within lib/index.js, we add a new method to the prototype of Mp3dat. Instead of taking a filename for the first parameter, it will accept an object (which we'll call opts) that must contain stream and size properties:

```
Mp3dat.prototype.statStream = function (opts, cb) {
  var self = this,
    errTxt = 'First arg must be options object with stream and size',
    validOpts = ({}).toString.call(opts) === '[object Object]'
    && opts.stream
    && opts.size
    && 'pause' in opts.stream
    && !isNaN(+opts.size);

  lib
  if (!validOpts) {
    cb(new Error(errTxt));
    return;
  }

  self.size = opts.size;
  self.f = opts.stream.path;

  self.stream = opts.stream;

  self._findBitRate(function (err, bitrate) {
    if (err) { cb(err); return; }
    self._buildStats(cb);
  });
}
```

Finally, just a few modifications to _findBitRate and we're done.

```
Mp3dat.prototype._findBitRate = function(cb) {
  var self = this,
    stream = self.stream || fs.createReadStream(self.f);
```

```

stream.on('data', function (data) {
  var i = 0;
  for (i; i < data.length; i += 2) {
    if (data.readUInt16LE(i) === 64511) {
      self.bitrate = self._bitrates[data.toString('hex', i + 2, i
+ 3)[0]];
      this.destroy();
      cb(null);
      break;
    };
  // rest of the _findBitRate function...
}

```

We conditionally hook onto either a passed in stream, or we create a stream from a given filename.

Let's run our tests (from the mp3dat folder):

```
node tests
```

The result should be:

```

passed
passed

```

One for stat, one for stat Stream

How it works...

We were already using a stream to retrieve our data. We simply expose this interface to the user by modifying `_findBitRate` so it either generates its own stream from a filename, or if a stream is present in the parent constructors properties (`self.stream`), it simply plugs that stream into the process that was already in place.

We then make this functionality available to the module user by defining a new API method: `statStream`. We conceptualize this first by making tests for it, then define it through `Mp3dat.prototype`.

The `statStream` method is similar to the `stat` method (in fact, we could merge them, see *There's more...*). Aside from checking the validity of the input, it simply adds one more property to an `Mp3dat` instance: the `stream` property, which is taken from `opts.stream`. For convenience, we cross reference `opts.stream.path` with `self.f` (this may or may not be available depending on the type of stream). This is essentially redundant but may be useful for debugging purposes on the users part.

At the top of `statStream` we have the `validOpts` variable, which has a series of expressions connected by `&&` conditionals. This is shorthand for a bunch of `if` statements. If any of these expression tests fail, the `opts` object is not valid. One expression of interest is '`pause`' in `opts.stream`, which tests whether `opts.stream` is definitely a stream or inherited from a stream (all streams have a `pause` method, and `in` checks for the property throughout the entire prototype chain). Another noteworthy expression among the `validOpts` tests is `!isNaN(+opts.size)`, which checks whether `opts.size` is a valid number. The `+` which precedes it converts it to a `Number` type and `isNaN` checks that it isn't "not a number" (there is no `isNumber` in JavaScript so we use `isNaN`).

There's more...

Now we have this new method. Let's write some more examples. We'll also see how we can merge `statStream` and `stat` together, and further enhance our module by causing it to emit events.

Making the STDIN stream example

To demonstrate usage with other streams we might write an example using the `process.stdin` stream as follows:

```
// to use try :  
// cat ../test/test.mp3 | node stdin_stream.js 82969  
// the argument (82969) is the size in bytes of the mp3  
  
if (!process.argv[2]) {  
  process.stderr.write('\nNeed mp3 size in bytes\n\n');  
  process.exit();  
}  
  
var mp3dat = require('../');  
process.stdin.resume();  
mp3dat.statStream({stream: process.stdin, size: process.argv[2]},  
function (err, stats) {  
  if (err) { console.log(err); }  
  console.log(stats);  
});
```

We've included comments in the example to ensure our users understand how to use it. All we do here is receive the `process.stdin` stream and the file size, then pass them to our `statStream` method.

Making the PUT upload stream example

In the *Handling file uploads* recipe of Chapter 2, *Exploring the HTTP Object*, we created a PUT upload implementation in the *There's more...* section of that recipe.

We'll take the `put_upload_form.html` file from that recipe, and create a new file called `HTTP_PUT_stream.js` in our `mp3dat/examples` folder.

```
var mp3dat = require('../..../mp3dat');
var http = require('http');
var fs = require('fs');
var form = fs.readFileSync('put_upload_form.html');
http.createServer(function (req, res) {
  if (req.method === "PUT") {
    mp3dat.statStream({stream: req, size: req.headers['content-length']}, function (err, stats) {
      if (err) { console.log(err); return; }
      console.log(stats);
    });
  }
  if (req.method === "GET") {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end(form);
  }
}).listen(8080);
```

Here, we create a server that serves the `put_upload_form.html` file. The HTML file allows us to specify a file to upload (which must be a valid MP3 file), and then sends it to the server.

In our server, we pass `req` (which is a stream) to the `statStream` property and `req.headers['content-length']` which gives us the size of MP3 in bytes as specified by the browser via the `Content-Length` header.

We then finish by logging `stats` to the console (we could also extend this example by sending `stats` back to the browser in JSON form).

Merging stat and statStream

There's a lot of similar code between `stat` and `statStream`. With a bit of restructuring, we can merge them into one method, allowing the user to pass either a string containing a filename or an object containing `stream` and `size` properties straight into the `stat` method.

First, we'd need to update our tests and examples. In `test/index.js`, we should remove the following code:

```
mp3dat.should.have.property('statStream');
mp3dat.statStream.should.be.an.instanceof(Function);
```

Since we're merging `stat` at `Stream` into `stat`, our two calls to `stat` and `stat Stream` should become:

```
mp3dat.stat({stream: fs.createReadStream(testFile),  
             size: fs.statSync(testFile).size}, cb);  
mp3dat2.stat(testFile, cb);
```

The `stat Stream` line in `examples/stdin_stream.js` should become:

```
mp3dat.stat({stream: process.stdin, size: process.argv[2]})
```

In `HTTP_PUT_stream.js` it should be:

```
mp3dat.stat({stream: req, size: req.headers['content-length']})
```

In `lib/index.js`, we trash the `streamStat` method, inserting a `_compile` method:

```
Mp3dat.prototype._compile = function (err, fstatsOpts, cb) {  
  var self = this;  
  self.size = fstatsOpts.size;  
  self.stream = fstatsOpts.stream;  
  self._findBitRate(function (err, bitrate) {  
    if (err) { cb(err); return; }  
    self._buildStats(cb);  
  });  
}
```

Finally, we modify our `Mp3dat.prototype.stat` method as follows:

```
Mp3dat.prototype.stat = function (f, cb) {  
  var self = this, isOptsObj = ({}).toString.call(f) === '[object  
Object]';  
  
  if (isOptsObj) {  
    var opts = f, validOpts = opts.stream && opts.size  
      && 'pause' in opts.stream && !isNaN(+opts.size);  
    errTxt = 'First arg must be options object with stream and size'  
  
    if (!validOpts) { cb(new Error(errTxt)); return; }  
  
    self.f = opts.stream.path;  
    self._compile(null, opts, cb);  
    return;  
  }  
  
  self.f = f;  
  fs.stat(f, function (err, fstats) {  
    self._compile.call(self, err, fstats, cb);  
  });  
}
```

The code that actually generates the `stats` has been placed into the `_compile` method. If the first argument is an object, we assume a stream and `stats` take on the role of the former `statStream`, calling `_compile` and returning from the function early. If not, we assume a filename and invoke `_compile` within the `fs.stat` callback with JavaScript's `call` method, ensuring our `this / self` variable carries through the `_compile` method.

Integrating the EventEmitter

Throughout this book, we have generally received data from modules via callback parameters or through listening for events. We can extend our `modules` interface further, allowing users to listen for events by causing Node's `EventEmitter` to adopt our `Mp3dat` constructor.

We need to require the `events` and `util` modules, then hook up `Mp3dat` with `EventEmitter` by assigning the `this` object of `Mp3dat` to it, and then give it the super powers of `Mp3dat` `EventEmitter` by using `util.inherits`:

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

function Mp3dat() {
    if (!(this instanceof Mp3dat)) {
        return new Mp3dat();
    }
    EventEmitter.call(this);
    this.stats = {duration: {}};
}

util.inherits(Mp3dat, EventEmitter);
```

All we do now is go through the existing methods of `Mp3dat` and insert the `emit` events in relevant places. We can emit the `bitrate` once it's found as follows:

```
Mp3dat.prototype._findBitRate = function(cb) {
    // beginning of _findBitRate method
    for (i; i < data.length; i += 2) {
        if (data.readUInt16LE(i) === 64511) {
            self.bitrate = self._bitrates[data.toString('hex', i + 2,
                i + 3)[0]];
            this.destroy();
            self.emit('bitrate', self.bitrate);
            cb(null);
            break;
        };
    // rest of _findBitRate method
```

Where we would callback with an error, we can also emit that error as shown in the following code:

```
// last part of _findBitRate method
}).on('end', function () {
  var err = new Error('could not find bitrate, is this definately an
  MPEG-1 MP3?');
  self.emit('error', err);
  cb(err);
});
```

Then there's the time signature:

```
Mp3dat.prototype._timesig = function () {
  // _timesig function code....
  self.emit('timesig', ts);
  return ts;
}
```

And of course, the `stats` object:

```
Mp3dat.prototype._buildStats = function (cb) {
  // _buildStats code
  self._timeProcessor(hours, function (duration) {
    // _timeProcessor code
    self.emit('stats', self.stats);
    if (cb) { cb(null, self.stats); }
  });
}
```

We've also added `if (cb)` to `_buildStats`, since a callback may no longer be necessary if the user opts to listen for events instead.

If a module user is dynamically generating the `Mp3dat` instances, they may wish to have a way to hook into a spawned instance event:

```
Mp3dat.prototype.spawnInstance = function () {
  var m = Mp3dat();
  this.emit('spawn', m);
  return m
}
```

Finally, to allow chaining, we can also return the `Mp3dat` instance from the `stat` function from two places. First within the `isOptsObj` block as follows:

```
Mp3dat.prototype.stat = function (f, cb) {
  // stat code
```

```
if (isOptsObj) {
    // other code here
    self._compile(null, opts, cb);
    return self;
}
```

Then right at the end of the function, as shown in the following code:

```
// prior stat code
self.f = f;
fs.stat(f, function (err, fstats) {
    self._compile.call(self, err, fstats, cb);
});
return self;
}
```

This is because we return early from the function depending on the detected input (filename or stream), so we have to return `self` from two places.

Now we can write an example for our new user interface. Let's make a new file in `mp3dat/examples` called `event_emissions.js`.

```
var mp3dat = require('../index');

mp3dat
  .stat('../test/test.mp3')
  .on('bitrate', function (bitrate) {
    console.log('Got bitrate:', bitrate);
  })
  .on('timesig', function (timesig) {
    console.log('Got timesig:', timesig);
  })
  .on('stats', function (stats) {
    console.log('Got stats:', stats);
    mp3dat.spawnInstance();
  })
  .on('error', function (err) {
    console.log('Error:', err);
  })
  .on('spawn', function (mp3dat2) {
    console.log('Second mp3dat', mp3dat2);
  });

```

See also

- ▶ *Creating a test-driven module API* discussed in this chapter
- ▶ *Handling file uploads* discussed in *Chapter 2, Exploring the HTTP Object*
- ▶ *Deploying a module to npm* discussed in this chapter

Deploying a module to npm

Now that we've created a module, we can share it with the rest of the world using the same integrated tool that we retrieve modules with: npm

Getting ready

Before we can deploy to npm we need to make a `package.json` file, so let's do that for our module. In `mp3dat`, we'll create `package.json` and add the following code:

```
{  
  "author": "David Mark Clements <contact@davidmarkclements.com>  
(http://davidmarkclements.com)",  
  "name": "mp3dat",  
  "description": "A simple MP3 parser that returns stat infos in a  
similar style to fs.stat for MP3 files or streams. (MPEG-1 compatible  
only)",  
  "version": "0.0.1",  
  "homepage": "http://nodecookbook.com/mp3dat",  
  "repository": {  
    "type": "git",  
    "url": "git://github.com/davidmarkclements/mp3dat.git"  
  },  
  "main": "./lib/index.js",  
  "scripts": {  
    "test": "node test"  
  },  
  "engines": {  
    "node": "~0.6.13"  
  },  
  "dependencies": {},  
  "devDependencies": {}  
}
```

We can of course insert our own name and the name of the package. Another way to create a `package.json` file is to use `npm init` which asks a series of questions via the command line, then generates the `package.json` file.

We can specify a repository in `package.json`. It's a good idea to use an online repository such as GitHub to manage version control, share code, and allow others to work on your code. See `http://help.github.com` to get started.

The `main` property is important. It defines the entry point to our module, which in our case is `./lib/index.js` (although we could have specified `./index.js` which loads `./lib/index.js`). By defining `scripts.test` as `node test`, we can now run `npm test` (or `npm np3dat test` once `np3dat` is installed via `npm`) to execute our unit tests.

We'll be deploying our module to `npm` as we left it in the previous recipe, where `stat` and `statStream` were both merged into `stat`, and we have integrated with `EventEmitter`.

How to do it...

To deploy to `npm`, we must have a developer account. We do this by executing the following:

```
npm adduser
```

And filling in our desired username, password, and contact email. That's it, we are now registered.

Before we go ahead and publish our module, we'll want to test that `npm` will install it on our system without a hitch. Inside `np3dat` we do the following:

```
sudo npm install -g
```

Then if we run `node` from the command line, we should be able to:

```
require('np3dat')
```

Without getting an error message. If it worked, we can go ahead and publish our module! Within `np3dat` we say the following:

```
npm publish
```

Now if we go to a completely different folder (say our home folder) and type the following:

```
npm uninstall mp3dat
npm install mp3dat
```

`npm` should install our package from its repository.

We can double check if it's there with the following command:

```
npm search mp3dat
```

Or, if that's taking too long, we can go to `http://search.npmjs.org/` in our browser. Our module will probably be on the home page (which contains the most recently published modules). Or we can hit `http://search.npmjs.org/#/mp3dat` to head to our module's `npm` registry page directly.

How it works...

npm is a command-line script written in Node which provides some excellent tools for developing and publishing modules. The tools really do what they say on the tin, add user adds a user, install installs, and publish publishes. It's really very elegant.

On the server side, the npm registry is backed by a CouchDB database that holds all the JSON-like data for each package. There's even a CouchDB _changes field we could hook into. On the command line, we could do:

```
curl http://isaacs.couchone.com/registry/_changes?feed=continuous&include_docs=true
```

And watch modules as they are added and modified in real time. If nothing is happening, we could always open another terminal and type the following commands:

```
mp3dat unpublish  
mp3dat publish
```

Which will cause the CouchDB changes feed to update.

There's more...

npm has some really nice features, let's take a look at some.

npm link

The npm link command can be useful for module authors.

Throughout development, if we wanted to require mp3dat as a global module, for example as require('mp3dat'), each time we make changes we could update the global package by running:

```
sudo npm install . -g
```

However, npm link provides an easier solution, when we run the following command:

```
sudo npm link
```

Within our mp3dat folder, a symlink is created from our global node_modules folder to our working directory. This causes Node to treat mp3dat as an installed global module, but any changes we make to our development copy will be reflected globally. When we are finished developing, and want to freeze the module on our system, we simply unlink as follows:

```
sudo npm unlink -g mp3dat
```

.npmignore and npm version

Our `example` files may be handy on GitHub, but we may decide they're of little benefit within npm. We can use an `.npmignore` file to keep certain files out of published npm packages. Let's create `.npmignore` in the `mp3dat` folder, and put:

```
examples/
```

Now, when we re-publish to the npm registry, our new package will be minus the `examples` folder. Before we can publish though, we either have to unpublish, or change the version of our package (or we could use the `--force` argument). Let's change the version, then publish again:

```
npm version 0.0.2 --message "added .npmignore"  
npm publish
```

Changing the version will also alter our `package.json` file to the new version number.

See also

- ▶ *Writing a functional module mock-up* discussed in this chapter
- ▶ *Refactoring from functional to prototypical* discussed in this chapter
- ▶ *Extending the module's API* discussed in this chapter
- ▶ *Accessing CouchDB changes stream with Cradle* discussed in *Chapter 4, Interfacing with Databases*

10

Taking It Live

In this chapter we will cover:

- ▶ Deploying to a server environment
- ▶ Automatic crash recovery
- ▶ Continuous deployment
- ▶ Hosting with a Platform as a Service provider

Introduction

Node is an excellent platform of choice for constructing and providing online services. Whether it's a simple, lean website, a highly versatile web app, or services that transcend beyond HTTP, at some point we must deploy our creations.

This chapter focuses on what it takes to bring our Node apps live.

Deploying to a server environment

Virtual Private Servers (VPS), Dedicated Servers, or Infrastructure as a Service (IaaS for example, the likes of Amazon EC2 or Rackspace) and owning our own server machines all have one thing in common: total control over the server environment.

However, with great power comes great responsibility, and there are a few challenges we need to be aware of. This recipe will demonstrate how to overcome these challenges as we safely initialize a Node web app on port 80.

Getting ready

We will, of course, need a remote server environment (or our own setup). It's important to research the best package for our needs.

Dedicated Servers can be expensive. The hardware to software ratio is one to one, we're literally renting a machine.

VPS can be cheaper since they share the resources of a single machine (or cluster), so we're only renting out the resources it takes to host an instance of an operating system. However, if we begin to use resources beyond those assigned, we could hit penalties (downtime, excessive charges) since over usage can affect other VPS users.

IaaS can be relatively cheap, particularly when up-scaling is involved (when we need more resources), though IaaS tends to contain a pay-as-you-go element to its pricing which means the costs aren't fixed and could require extra monitoring.

Our recipe assumes the usage of a Unix/Linux server with the `sshd` (SSH Service) running. Furthermore, we should have a domain pointed at our server. In this recipe, we'll assume the domain name as `nodecookbook.com`. Finally, we must have Node installed on our remote server. If difficulties arise, we can use the instructions available at <https://www.github.com/joyent/node/wiki/Installation>, or for installing via a package manager we can use the instructions at <https://www.github.com/joyent/node/wiki/Installing-Node.js-via-a-package-manager>.

We'll be deploying the `login` app from the second-to-last recipe of *Chapter 6, Accelerating Development with Express*, so we need this at hand.

How to do it...

To ready our app for transfer to the remote server, we'll remove the `node_modules` folder (we can rebuild it on the server):

```
rm -fr login/node_modules
```

Then we compress the `login` directory by executing the following:

```
npm pack login
```

This will generate a compressed archive named after the app's name and version as given in the `package.json` file, which will generate the filename `application-name-0.0.1.tgz` for an untouched Express generated `package.json` file.

Whatever `npm pack` called it, let's rename it to `login.tgz`:

```
mv application-name-0.0.1.tgz login.tgz #Linux/Mac OS X  
rename application-name-0.0.1.tgz login.tgz ::Windows.
```

Next, we upload `login.tgz` to our server. For example, we could use SFTP:

```
sftp root@nodecookbook.com
```

Once logged in to the via SFTP, we can issue the following commands:

```
cd /var/www  
put login.tgz
```

It's not necessary to upload to the `/var/www` directory, it's just a natural place to put a website.

This assumes that we have SFTPed into our server from the directory holding `login.tgz`.

Next, we SSH into the server:

```
ssh -l root nodecookbook.com
```



If we're using a Windows desktop, we could SFTP and SSH into our server using putty: `http://www.chiark.greenend.org.uk/~sgtatham/putty/`.



Once logged in to the remote server, we navigate to `/var/www` and decompress `login.tar.gz`:

```
tar -xvf login.tar.gz
```

As `login.tar.gz` decompresses, it recreates our `login` folder on the server.

To rebuild the `node_modules` folder, we enter the `login` folder and use `npm` to regenerate the dependencies.

```
cd login  
npm -d install
```

Most servers have a shell-based editor, such as `nano`, `vim` or `emacs`. We can use one of these editors to change one line in `app.js` (or otherwise SFTP over a modified `app.js`):

```
app.listen(80, function () { process.setuid('www-data'); });
```

We're now listening on the standard HTTP port, meaning we can access our app without suffixing a port number to its web address. However, since we'll be starting the app as `root` (necessary in order to bind to port 80), we also pass a callback to the `listen` method which changes access privileges of the app from `root` to `www-data`.

In some cases, dependent upon file permissions, reading or writing to files from our app may no longer work. We can fix this by changing ownership:

```
chown -R www-data login
```

Finally, we can start our app with:

```
cd login  
nohup node app.js &
```

We can ensure that our app is running as www-data with:

```
ps -ef | grep node
```

How it works...

We modified `app.listen` to bind to port 80 and added a callback function that resets the user ID from `root` to `www-data`.

Adding a callback to `listen` isn't limited to Express, it works the same way with a plain `httpServer` instance.

Running a web server as `root` is bad practice. If our app was compromised by an attacker, they would have `root` access to our system via our app's privileged status.

To demote our app, we call `process.setuid` and pass in `www-data`. `process.setuid`. This takes either the name of a user, or the user's UID. By passing in a name, we cause `process.setuid` to block the event loop (essentially freezing operations) while it cross-references the user string to its UID. This eliminates the potential sliver of time where the app is bound to port 80 and also running as `root`. In essence, passing a string to `process.setuid` instead of the underlying UID means nothing can happen until the app is no longer `root`.

We call our process with `nohup` and follow up with the ampersand (`&`). This means we freely end our SSH session without causing our app to terminate along with the session.

The ampersand turns our process into a background task, so we can do other things (like `exit`) while it runs. `nohup` means ignore the hangup signal (HUP). HUP is sent to any running processes initiated via SSH whenever the SSH session is terminated. Essentially, using `nohup` allows our web app to outlive the SSH session.

There's more...

There are other ways to start our app independent from our session, and to bind to port 80 without running the app as `root`. Plus, we can also run multiple apps and proxy them to port 80 with `http-proxy`.

Using screen instead of nohup

An alternative to using `nohup` to achieve independence from our SSH session is `screen`. We would use it as follows:

```
screen -S myAppName
```

This would give us a virtual terminal, from which we could say:

```
cd login  
node app.js
```

Then we could leave the virtual terminal by pressing `Ctrl + A` followed by `D`. We would return to our initial terminal. The virtual terminal would continue to run after we had logged out of SSH. We could also log back in to SSH at any time and say:

```
screen -r myAppName
```

Where we would be able to see any console output and stop (`Ctrl + C`) and start the app.

Using authbind for privileged ports

For this example, we should SSH into our server as a non-root user:

```
ssh -l dave nodecookbook.com
```

An alternative way to bind to port 80 is with `authbind`, which can be installed via our server's package manager. For instance, if our package manager is `apt-get` we could say:

```
sudo apt-get install authbind
```

`authbind` works by preempting the operating system policies on port binding and exploiting an environment variable called `LD_PRELOAD` upon execution. Therefore, it never needs to be run with `root` privileges.

To get it working for us we have to perform some simple configuration work as follows:

```
sudo touch /etc/authbind/byport 80  
sudo chown dave /etc/authbind/byport 80  
sudo chmod 500 /etc/authbind/byport 80
```

This tells `authbind` to allow the user `dave` to bind processes to port 80.

We no longer need to change the process UID, so we edit the penultimate line of `app.js` to:

```
app.listen(80);
```

We should also change ownership of the `login` folder as follows:

```
chown -R dave login
```

Now we can start our server without touching the `root` access at all:

```
nohup authbind node app.js &
```

`authbind` can cause our app to work out of the box, no modifications necessary. However, it currently lacks IPv6 support so it's not yet future-proof.

Hosting multiple processes from port 80

What about serving multiple processes with the default HTTP port?

We can achieve this with the third-party `http-proxy` module.

```
npm install http-proxy
```

Let's say we have two apps one (our `login` app) to be hosted at `login.nodecookbook.com` and the other (the `server.js` file from the very first recipe of this book) to be simply at `nodecookbook.com`. Both domains point to the same IP.

`server.js` will be listening on port 8080, and we'll modify `login/app.js` to listen again to port 3000 as shown in the following code:

```
app.listen(3000, '127.0.0.1');
```

We also added a second argument defining what address to bind to (rather than any address). This prevents our server from being accessed by port.

Let's create a file in a new folder, call it `proxy.js`, and write the following:

```
require('http-proxy')
.createServer({router: {
  'login.nodecookbook.com': 'localhost:3000',
  'nodecookbook.com': 'localhost:8080'
}}).listen(80, function () {
  process.setuid('www-data');
});
```

The object passed to `createServer` contains a `router` property, which in turn is an object instructing `http-proxy` to route incoming traffic on a particular domain to the correct locally-hosted process according to its port.

We finish off by binding to port 80 and degrading from `root` to `www-data`.

To initialize, we must do:

```
nohup node login/app.js &
nohup node server.js &
nohup node proxy.js &
```

Since we're binding our proxy server to port 80, these commands must be run as `root`. If we're operating SSH with a non-root account, we simply prefix these three commands with `sudo`.

See also

- ▶ *Automatic crash recovery* discussed in this chapter
- ▶ *Continuous deployment* discussed in this chapter
- ▶ *Hosting with a Platform as a Service provider* discussed in this chapter

Automatic crash recovery

When we create a site, server and site logic is all tied up in one process. Whereas with other platforms, the server code is already in place. If our site code has bugs, the server is very unlikely to crash, and thus in many cases the site can stay active even if one part of it is broken.

With a Node-based website, a small bug can crash the entire process, and this bug may only be triggered once in a blue moon.

As a hypothetical example, the bug could be related to character encoding on POST requests. When someone like Felix Geisendörfer completes and submits a form, suddenly our entire server crashes because it can't handle umlauts.

In this recipe, we'll look at using Upstart, an event-driven init service available for Linux servers, which isn't based upon Node, but is nevertheless a very handy accomplice.

Getting ready

We will need Upstart installed on our server. <http://upstart.ubuntu.com> contains instructions on how to download and install. If we're already using an Ubuntu or Fedora remote server then Upstart will already be integrated.

How to do it...

Let's make a new server that purposefully crashes when we access it via HTTP:

```
var http = require('http');
http.createServer(function (req, res) {
  res.end("Oh oh! Looks like I'm going to crash... ");
  throw crashAhoy;
}).listen(8080);
```

After the first page loads, the server will crash and the site goes offline.

Let's call this code `server.js` placing it on our remote server under
`/var/www/crashingserver`

Now we create our Upstart configuration file, saving it on our server as `/etc/init/crashingserver.conf`.

```
start on started network-services

respawn
respawn limit 100 5

setuid www-data

exec /usr/bin/node /var/www/crashingserver/server.js >> \ /var/log/
crashingserver.log 2>&l

post-start exec echo "Server was (re)started on $(date)" | mail -s
"Crashing Server (re)starting" dave@nodecookbook.com
```

Finally, we initialize our server as follows:

```
start crashingserver
```

When we access `http://nodecookbook.com` 8080 and refresh the page, our site is still accessible. A quick look at `/var/log/crashingserver.log` reveals that the server did indeed crash. We could also check our inbox to find the server restart notification.

How it works...

The name of the Upstart service is taken from the particular Upstart configuration filename. We initiate the `/etc/init/crashingserver.conf` Upstart service with `start crashingserver`.

The first line of the configuration ensures our web server automatically recovers even when the operating system on our remote server is restarted (for example, due to a power failure or required reboot, and so on).

`respawn` is declared twice, once to turn on respawning and then to set a `respawn_limit` — a maximum of 100 restarts every 5 seconds. The limit must be set according to our own scenario. If the website is low traffic this number might be adjusted to say 10 restarts in 8 seconds.

We want to stay alive if at all possible, but if an issue is persistent we can take that as a red flag that a bug is having a detrimental effect on user experience or system resources.

The next line initializes our server as the `www-data` user, and sends output to `/var/log/crashingserver.log`.

The final line sends out an email just after our server has been started, or restarted. This is so we can be notified that there are probably issues to address with our server.

There's more...

Let's implement another Upstart script that notifies us if the server crashes beyond its `respawn limit`, plus we'll look at another way to keep our server alive.

Detecting a respawn limit violation

If our server exceeds the `respawn limit`, it's likely there is a serious issue that should be solved as soon as possible. We need to know about it immediately. To achieve this in Upstart, we can create another Upstart configuration file that monitors the `crashingserver` daemon, sending an email if the `respawn limit` is transgressed.

```
task

start on stopped crashingserver PROCESS=respawn

script
  if [ "$JOB" != '' ]
    then echo "Server \"$JOB\" has crashed on $(date)" | mail -s \
      "$JOB" site down!!" dave@nodecookbook.com
  fi
end script
```

Let's save this to `/etc/init/sitedownmon.conf`.

Then we do:

```
start crashingserver
start sitedownmon
```

We define this Upstart process as a task (it only has one thing to do, after which it exits). We don't want it to stay alive after our server has crashed.

The task is performed when the `crashingserver` daemon has stopped during a respawn (for example, when the `respawn limit` has been broken).

Our script stanza (directive) contains a small bash script that checks for the existence of the `JOB` environment variable (in our case, it would be set to `crashingserver`) and then sends an email accordingly. If we don't check its existence, a `sitedownmon` seems to trigger false positives when it is first started and sends an email with an empty `JOB` variable.

We could later extend this script to include more web servers, simply by adding one line to `sitedownmon.conf` per server:

```
start on stopped anotherserver PROCESS=respawn
```

Staying up with forever

There is a simpler Node-based alternative to Upstart called `forever`:

```
npm -g install forever
```

If we simply initiate our server with `forever` as follows:

```
forever server.js
```

And then access our site, some of the terminal output will contain the following:

```
warn: Forever detected script exited with code: 1
warn: Forever restarting script for 1 time
```

But we'll still be able to access our site (although it will have crashed and been restarted).

To deploy our site on a remote server, we log in to our server via SSH, install `forever` and say:

```
forever start server.js
```

While this technique is certainly less complex, it's also less robust. Upstart provides core kernel functionality and is therefore system critical. If Upstart fails, the kernel panics and the whole server restarts.

Nevertheless, `forever` is used widely in production on Nodejitsu's PaaS stack, and its attractive simplicity may be viable for less mission-critical production environments.

See also

- ▶ *Deploying to a server environment* discussed in this chapter
- ▶ *Hosting with a Platform as a Service provider* discussed in this chapter
- ▶ *Continuous deployment* discussed in this chapter

Continuous deployment

The more streamlined our processes, the more productive we can be. Continuous deployment is about committing small ongoing improvements to a production server in a time saving, efficient way.

Continuous deployment is especially relevant to team collaboration projects. Instead of working on separate forks of the code and spending extra time, money, and effort on integration, everyone works on the same code base so integration is seamless.

In this recipe, we'll create a deployment flow using Git as a version control tool. While this may not be Node, it can certainly boost productivity for coding, deploying, and managing Node projects.



If we're a little unfamiliar with Git, we can gain insight from Github's help documents, <http://help.github.com>



Getting ready

We'll need Git installed on both our server and desktop systems, instructions for different systems can be found here http://book.git-scm.com/2_installing_git.html. If we're using Linux with the `apt-get` package manager we can do:

```
sudo apt-get install git git-core
```

If we are installing Git for the first time, we'll have to set the personal information configuration settings as follows:

```
git config --global user.name "Dave Clements"  
git config --global user.email "dave@nodecookbook.com"
```

We'll be using our `login` app, which we deployed to our server in the first recipe. So let's SSH into our server and enter the `/var/www/login` directory.

```
ssh -l root nodecookbook.com -t "cd /var/www/login; bash"
```

Since we'll not be running our app as root, we'll keep things simple and change the listening port in `login/app.js` to 8000:

```
app.listen(8000);
```

How to do it...

Once we've logged in to our server and installed Git (see *Getting ready*) in the `login` folder, we say the following:

```
git init  
git add *  
git commit -m "initial commit"
```

Next, we create a bare repository (it has a record of all the changes but no actual working files) which we'll be pushing changes to. This helps to keep things consistent.

Taking It Live

We'll call the bare repository `repo`, because this is the repository we'll be pushing our changes to and we'll create it within the `login` folder:

```
mkdir repo
echo repo > .gitignore
cd repo
git --bare init
```

Next, we hook up our bare `repo` to the `login` app repository, and push all the commits to `repo`.

```
cd ..
git remote add repo ./repo
git push repo master
```

Now we'll write a Git hook that instructs the `login` repository to pull any changes from the bare `repo` repository, then restarts our `login` app whenever `repo` is updated via a remote Git push.

```
cd repo/hooks
touch post-update
chmod +x post-update
nano post-update
```

With the file open in `nano`, we write the following code:

```
#!/bin/sh

cd /root/login
env -i git pull repo master

exec forever restart /root/login/app.js
```

Saving our hook with `Ctrl + O`, then exit with `Ctrl + X`.

If we ever make Git commits to the `login` repository, the two repositories could go out of sync. To fix this, we create another hook for the `login` repository:

```
#!/bin/sh
git push repo
```

We store this in `login/.git/hooks/post-commit`, ensuring it has been made executable using `chmod +x post-commit`.

We'll be making commits to the `repo` remotely via the SSH protocol. Ideally, we want to create a system user just for Git interactions.

```
useradd git  
passwd git      #set a password  
  
mkdir /home/git  
chown git /home/git
```

We've also created home directories for the `git` user to make it easy for `forever` to store logs and PID files. We'll need to make `git` the owner of the `login` app, allowing us to manage it using Git through SSH:

```
cd /var/www  
chown -R git login
```

Finally (for the server-side setup), we log in as the `git` user and start our app using `forever`.

```
su git  
forever start /var/www/login/app.js
```

Assuming our server is hosted at `nodecookbook.com`, we could now access the `login` app at `http://nodecookbook.com:8000`.

Back on our desktop, we clone the `repo` repository:

```
git clone ssh://git@nodecookbook.com/var/www/login/repo
```

This will give us a `repo` directory, containing all the generated files perfectly matching our original `login` folder. We can then enter the `repo` folder and make a change to our code (say, altering the port in `app.js`).

```
app.listen(9000);
```

Then we commit the change and push to our server.

```
git commit -a -m "changed port"  
git push
```

On the server side, our app should have automatically restarted, resulting in our app now being hosted from `http://nodecookbook.com:9000` instead of `http://nodecookbook.com:8000`.

How it works...

We created two Git repositories. The first is the `login` app itself. When we ran `git init`, a `.git` directory was added to the `login` folder. `git add *` adds all of the files in the folder and `commit -m "initial commit"` plants our additions into Git's version control system. So now our entire code base is recognized by Git.

The second is `repo`, which is created with the `--bare` flag. This is a sort of skeleton repository providing all of the expected Git functionality, but lacking the actual files (it has no working tree).

While it may seem overly complex to use two repositories, it actually simplifies things greatly. Since Git does not allow pushes to a branch that is currently checked in, we would have to create a separate dummy branch so we can checkout of the master and into the dummy branch. This creates problems with the Git hooks and restarting our app. The hooks try to start the app for the wrong branch. The branches can also quickly become out of sync, and the hooks only add fuel to the fire.

As `repo` is within the `login` directory, we create a `.gitignore` file telling Git to disregard this subdirectory. Even though `login` and `repo` are on the same server, we add `repo` as a `remote` repository. This puts some necessary distance between the repositories and allows us to later use our first Git hook to cause `login` to pull changes from `repo`. A Git push from `repo` to `login` wouldn't cause `login` to update its working directory, whereas pulling from `repo` into `login` does initiate a merge.

After our `remote add`, we perform an initial push from the master branch (`login`) to `repo`, now they're singing off the same hymn sheet.

Then we created our hooks.

Git hooks are executable files which reside in the repository's hook folder. There are a variety of available hooks (already in the folder, but suffixed with `.sample`). We used two: `post-update` and `post-commit`. One executes after an update (for example, once changes have been pulled and integrated into `repo`), and one after a commit.

The first hook, `login/repo/hooks/post-update`, essentially provides our continuous deployment functionality. It changes its working directory from `repo` to `login` using `cd`, and commands a `git pull`. The `git pull` command is prefixed with `env -i`. This prevents problems with certain Git functionality that would otherwise execute the Git commands on behalf of `repo` no matter what directory we sent our hook script to. Git utilizes a `$GIT_DIR` environment variable to lock us in to the repository that the hook is called from. `env -i` deals with this by telling `git pull` to ignore (`-i`) all environment variables.

Having updated the working directory, our hook then goes on to call `forever restart`, thus causing our app to reinitialize with the committed changes in place.

Our second hook is little more than a polyfill to ensure code base consistency in the event that commits are made directly to the `login` repository. Making commits directly to the `login` directory won't update the working tree, nor will it cause our app to restart but the code between `login` and `repo` will at least maintain synchronicity.

For the sake of damage limitation (if we were ever compromised), we create a specific account for handling Git updates over SSH, giving it a home directory, taking ownership of the `login` app and executing the primary initialization of our app.

Once the server is configured it's plain sailing. After cloning the `repo` repository to our local development environment, we simply make a change, add and commit that change, then push to the server.

The server receives our push request, updates `repo`, initiates the `post-update` hook which makes `login` pull the changes from `repo`, after which the `post-update` hook uses forever to restart `app.js`, and thus we have a continuous deployment work flow.

We can potentially have as many clones from as many locations as we like, so this method lends itself well to geographically-independent team collaboration projects both large and small.

There's more...

We could avoid uploading modules by using `npm install` in the post-update hook. Also, Git hooks don't have to be written in shell script, we can write them in Node!

Building module dependencies on update

Some Node modules are written in pure JavaScript, others have C or C++ bindings. Those with C or C++ bindings have to be built from source—a task which is system specific. Unless our live server environment is identical to our development environment, we shouldn't simply push code build for one system onto another.

Further, to save on transfer bandwidth and have faster synchronizations, we could have our Git hooks install all modules (native bindings and JavaScript) and have Git ignore the `node_modules` folder entirely.

So in our local repository, let's do the following:

```
echo node_modules >> .gitignore
```

Then we'll change the `post-update` hook in our bare remote repository (`login/repo/hooks`) to:

```
#!/bin/sh

cd /root/login

env -i git pull repo master && npm rebuild && npm install

exec forever restart /root/login/app.js
```

We've added `&& npm rebuild && npm install` to the `git pull` line (using `&&` to ensure they benefit from the `env -i` command).

Now if we added a module to `package.json`, and did a `git commit -a` followed by `git push`, our local `repo` would push the `package.json` to the remote `repo`. This would trigger the `post-update` hook to pull changes into the main `login` repository, and follow this up with an `npm rebuild` (to rebuild any C / C++ dependencies) and an `npm install` (to install any new modules).

Writing a Node Git hook for integrated testing

Continuous deployment is an extension of continuous integration which generally carries the expectation that a thorough test suite is run against any code changes for quality assurance.

Our `login` app (being a basic demonstration site) doesn't have a test suite (for info on test suites, see *Chapter 9, Writing Your Own Node Modules*), but we can still write a hook that executes any tests that could be added to `login` in the future.

What's more, we can write it in Node, which has the added bonus of functioning cross platform (on Windows, for example).

```
#!/usr/bin/env node

var npm = require("npm");

npm.load(function (err) {
  if (err) { throw err; }

  npm.commands.test(function (err) {
    if (err) { process.exit(1); }
  });
});
```

We would place this code on the server into `logi n/ r epos/ hooks/ pr e- c omm i t` and make it executable (`chmod +x pr e- c omm i t`).

The first line sets `node` as the scripts interpreter directive (much as `#!/ bin/ sh` sets the `sh` shell for shell scripts). Now we're in Node country.

We use `npm` programmability, to load the `package.json` file for our app, and then run the test script (if any is specified).

We then add the following to our `package.json` file:

```
{  
  "name": "application-name"  
, "version": "0.0.1"  
, "private": true  
, "dependencies": {  
    "express": "2.5.5"  
, "jade": ">= 0.0.1"  
},  
  "scripts": {  
    "test": "node test"  
},  
  "devDependencies": {"npm": "1.1.18"}  
}
```

Then do the following:

```
npm -d install
```

Now whenever we push to `repo`, any changes will only be committed if they pass the tests. As long as we have a well-written test suite, this is a great way to maintain good code.



For our `scripts.test` property, we used `node test` (as in *Chapter 9, Writing Your Own Node Modules*). However, there are more advanced test frameworks available to us, such as `Mocha` <http://visionmedia.github.com/mocha/>.



This Node Git hook is adapted (with permission) from a `gist` by Domenic Denicola, which can be found at <https://gist.github.com/2238951>.

See also

- ▶ *Deploying to a server environment* discussed in this chapter
- ▶ *Automatic crash recovery* discussed in this chapter
- ▶ Creating a test-driven module API discussed in *Chapter 9, Writing Your Own Node Modules*
- ▶ *Hosting with a Platform as a Service provider* discussed in this chapter

Hosting with a Platform as a Service provider

A **Platform as a Service Provider (PaaS)** for Node incorporates all of the concepts discussed in the previous three chapters and boils deployment down to a very basic, yet powerful, set of commands. When it comes to deployment, PaaS can make our lives very easy. With one simple command our app is deployed, and with another we can seamlessly update and reinitialize.

In this example, we'll learn how to deploy to Nodejitsu, one of the leading Node hosting platform providers.

Getting ready

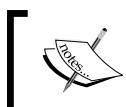
First, we'll install `jitsu`, Nodejitsu's deployment and app management command-line app.

```
sudo npm -g install jitsu
```

Before we proceed, we must sign up for an account as follows:

```
jitsu signup
```

The app will take us through the trivial signup process and create an account for us, which we must confirm by email.



Nodejitsu is not the only Node PaaS, there are others such as no.de, Nodester, and Cloud Foundry which follow similar processes.

Once we've received our email, we use the provided voucher, for instance:

```
jitsu users confirm davelements _sCjXz46in-6IBpI
```

As in the first recipe, we'll use the `login` app from the *Initializing and using a session* recipe of *Chapter 6, Accelerating Development with Express*.

How to do it...

First of all, we enter the `login` folder and make some modifications to `package.json`:

```
{  
  "name": "ncb-login",  
  "version": "0.0.1",  
  "private": true,  
  "dependencies": {  
    "express": "2.5.5",  
    "jade": ">= 0.0.1"  
  },  
  "subdomain": "login",  
  "scripts": {  
    "start": "app.js"  
  },  
  "engines": {  
    "node": "0.6.x"  
  }  
}
```

And now we deploy!

```
jitsu deploy
```

If we navigate to our specified subdomain at `http://login.nodejitsu.com` or alternatively `http://login.jitsu`, we will see our `login` app (if a subdomain isn't available `jitsu` will suggest alternatives).

How it works...

We made some modifications to `package.json`. The name of our app is the only alteration that is necessarily made by directly editing `package.json`. The other additions could have been made on our behalf by the `jitsu` executable. It is important to set the name of the app because in `jitsu` apps are managed by their name.

If we had not appended the `subdomain`, `scripts`, and `engines` properties to `package.json`, `jitsu` would have asked for the particulars when we ran `jitsu deploy` and regenerated `package.json` on our behalf.

`subdomain` specifies the label prefix to `nodejitsu.com`, from where we host our app (for example, `login.nodejitsu.com`). `scripts`, with the `start` sub-property informs Nodejitsu of our ignition script, the file that starts the app. `engines` defines which versions of Node our app is designed for.

There's more...

Let's find out how to access our Nodejitsu app via a custom domain, and how to provision a database backend through the `jitsu` executable.

Assigning custom domains to Nodejitsu apps

To prepare our app for serving through a custom domain, we make an amendment to `package.json` as shown in the following code:

```
// prior package.json data
"subdomain": "login",
"domains": "login.nodecookbook.com",
"scripts": {
  "start": "app.js"
},
// rest of package.json data
```

Then we push our changes with `jitsu` as follows:

```
jitsu apps update ncb-login
```

The app is now ready to receive traffic via `http://login.nodecookbook.com`, but before traffic can reach it, we must match our domain's A records with those of Nodejitsu.

We can get the current list of Nodejitsu A records with `dig` (or a similar command-line app):

```
dig nodejitsu.com
```

The process for changing A records depends upon our domain providers. We can generally find it in the DNS area of our provider's control panel/administration area.

Provisioning a database with `jitsu`

In the last recipe of *Chapter 6, Accelerating Development with Express*, we built a MongoDB backed Express app. Now we're going to take the `profiler` app live with Nodejitsu, making use of the database provisioning capabilities of `jitsu`.

So let's provision a Mongo database for the profiler database as follows:

```
jitsu databases create mongo profiler
```

`jitsu` will provision our database through a third-party database PaaS provider (in Mongo's case, PaaS provider is MongoHQ). The second-to-last line of output provides us with the MongoDB URI for our new database, which will look something like the following code:

```
info: Connection url: mongodb://nodejitsu:14dce01bda24e5fe53bbdaa8f2f6547b@flare.mongohq.com 10019/nodejitsub169742247544
```

So we update the second line of `profiler/tools/prepopulate.js` to:

```
client = mongo.db('mongodb://nodejitsu:14dce01bda24e5fe53bbdaa8f2f65  
47b@flame.mongohq.com:10019/nodejitsudb169742247544'),
```

Then we run it from the `profiler/tools` folder:

```
node prepopulate.js
```

This fills our remote database with profiles and login data.

We update our database URI in two other places `profiler/profiles.js` and `profiler/login/login.js`, in both places the second line is altered to say:

```
db = mongo.db('mongodb://nodejitsu:14dce01bda24e5fe53bbdaa8f2f6547b@  
flame.mongohq.com:10019/nodejitsudb169742247544'),
```

Finally, we type the following:

```
jitsu deploy
```

`jitsu` will ask us for certain settings (`subdomain`, `scripts.start`, and `engines`), we can just press *Enter* and stick with the defaults (unless of course `profiler.nodejitsu.com` is already taken, in which case we should choose a different URL).

`jitsu` will then deploy our app, we should be able to access it at `profiler.nodejitsu.com`

See also

- ▶ *Deploying to a server environment* discussed in this chapter
- ▶ *Automatic crash recovery* discussed in this chapter
- ▶ *Continuous deployment* discussed in this chapter

Index

Symbols

.npmignore 299
_method 179

A

addContext method 267
addProfile function 73
AJAX
 browser-server transmission 65-67
anypageAdmin function 158
App mounting
 about 201
 data flow 202
attacks 210
authenticate function 222

B

Basic Authentication standard
 about 210-212
 Express, using 212
 working 212
bind method 99
browser-server transmission
 serialized data, sending from client to server
 70-73
 working 68, 69
 via AJAX 65-67
bufferOffset property 47
bufferSize property 22
buildXml function 71

C

cache.clean method 24
changes method 109
checkAndSave function 101
chunks property 47
Cluster Of Unreliable Commodity Hardware.
 See **Couch**
config method 119
connect.bodyParser
 using, for POST data access 33, 34
content
 caching 15-17
 changes, detecting 18
 working 17
contents property 242
continuous deployment
 about 310, 311
 module dependencies, building on update
 315, 316
 Node Git hook, writing 316, 317
 steps 311-313
 working 314, 315
Couch 101
CouchDB
 data retrieving, Cradle used 102, 103
 data storing, Cradle used 99, 100
 scaling, BigCouch used 101
CouchDB changes stream
 accessing, Cradle used 108-110
 working 111
CouchDB HTTP interface
 exposing, to remote connections 108

Cradle
admin user, creating 105
modifying operations, locking 106, 107
using, for data retrieval from CouchDB 102, 103
using, for data storage to CouchDB 99, 100
working 101

crash recovery
about 307
alternative, using 310
respawn limit violation, detecting 309
steps 307, 308
Upstart service , working 308

Cross Site Request Forgery. *See* **CSRF**

CSRF
about 227
POST forms, auto-securing 231-233
preventing 227-230
working 231
XSS circumvention 231
XSS vulnerabilities, eliminating 234, 235

CSS engines, with Express
using 169-171
working 171

CSV file
CSV elements, customizing 84
manipulating, as stream 85
reading 84
working 83
writing to 82, 83

custom event **130**

D

data event **260**

dataReady event **260**

Digest Authentication
about 218
authenticated areas, logging out 221-223
implementing 218, 219
working 220

digest.response property **221**

download throttling
about 45-48
implementing 46

resumes, enabling from broken downloads 50
working 48, 49

duration property **271**

dynamicHelpers method **184**

dynamic routing
about 156, 157
Asterisks wildcards 159
optional routes 158, 159
route validation 158
working 157

E

email
alternative transport, using 240
attachments, sending 241, 242
createTransport method, working 240
HTML emails 241
nodemailer, using 238, 239
sending 238
sendmail, using 240

env -i command **316**

environments
applying 153, 154
defining 152
NODE_ENV, changing 155, 156
setting 154, 155
working 154

err parameter **100**

example code
downloading 6

exports.stat function **283**

Express
CSS engines 169
helpers 205
logger, using 206-208
Mixins 204, 205
Node website, benchmarking 206
route handling 202, 203
styles 205
user flow 206
views 203

Express scaffolding
about 148

apart app.js 150, 151
generating 148
routes/index.js 152
working 149

Express web app

App mounting 201
mounted login App. modifying 197-201
profiler app, modifying 194-197
working 201

F

features, MongoDB

aggregation 93, 94
indexing 93, 94
limit update 94, 95
modifiers update 94, 95
sort update 94, 95
upserting option 95

fileBegin event 37

fileName property 242

filesystem hacking exploits security

about 24-26
node-static 28
whitelisting 27
working 27

file uploads

about 34
filenames preserving, formidable used 37
formidable, using 37
handling 35, 36
POST data, accepting 37
via PUT 38-40
working 36

findBitRate function 281

forceLogOut property 223

forEach method 42

format property 208

fs.watch method 145

fullname property 53

functional module mock-up

use case example, writing 279
writing 275-278

G

generateTextFromHtml property 241

Github page 235

H

Hash-based Message Authentication Code.

See HMAC

helpers 205

HMAC 216

hotTrends.xmlHandler method 79

href attribute 234

HTTP 121

http module 29

HTTPS web server

about 224

Basic Authentication securing, SSL/TLS used 226

in Express 226

setting up 224, 225

working 225

I

IncomingForm class 36

INCR command 143

io.sockets connection event 142

J

Jade mixins 204

javascript

protocol 235

JavaScript Object Notation. See JSON

jitsu 321

JSON

about 51

JSONP responses, constructing 54, 55

object, converting to 52, 53

working 54

JSONP 54

JSON with Padding. See JSONP

L

LAMP **85**

Linux Apache MySQL PHP. See **LAMP**

M

Mailinator **239**

makeCall function **76**

makeCall method **248**

module

about 269

deploying, to npm 296-299

module's API

EventEmitter, integrating 293-295

extending 287, 288

PUT upload stream example, creating 291

stat, merging with statStream 291, 292

STDIN stream example, creating 290

working 289

MongoDB

data, retrieving 90-92

data, storing 90-92

features 93

working 92, 93

Mongoskin

binding, collecting 99

collection binding 98, 99

data, retrieving 96

data, storing 96

working 97

multipartAssembler function **45**

multipart message **34**

N

network latency

overcoming, by command pipelining 114, 115

network paradigms

integrating 237

Node

about 301

feature 5

using, as HTTP client 41

Node Git hook **317**

node-static module **28**

Node, using as HTTP client

about 40, 41

multipart file upload 43-45

POST requests, sending 42

working 41, 42

Nowjs **136-138**

npm **298**

npm link command **298**

npm version **299**

O

object

converting to JSON 51054

converting to XML 56-58

object conversion, to XML

about 56

arrays 59, 60

functions 59, 60

steps 56, 57

text values 63

text values 64

working 58

XML attributes, generating 61-63

P

PaaS

about 318

custom domains, assigning to Nodejitsu apps

320

database, provisioning 320, 321

hosting with 318, 319

working 319

password hashing

about 213, 214

crypto module 216

PBKDF2, using 216, 217

steps 214, 215

unifying, HMAC used 216

working 215

patchMixins function **197**

PBKDF2 **216, 217**

performance optimization

streaming, using 19-21

working 21, 22

Platform as a Service Provider. *See* **PaaS**

poison null byte attacks **27**

POST data
accessing, connect.bodyParser used 33, 34
processing 29-32
working 32

POST server
protecting 31

post-update hook **315**

process memory overruns
protecting against 22-24

profileName property **73**

prototypical inheritance
about 279
code, rewriting 280-283
multiple instances, allowing for 285
stat function, adding 284, 285
working 283, 284

PubSub
implementing, Redis used 115-117
Redis authentication 118
working 117, 118

R

randomBytes method **217**

real-time widget
about 138-141
scalability 143
WebSockets, as development tool 144, 145
working 141, 142

Redis
about 111
data, retrieving 112
data, storing 112
node module, speeding up 114
PubSub, implementing 115-117
securing, from external connections 119
working 113

relative directory traversal **27**

response.finished property **8**

router
querystring, parsing 9, 10
routing modules 10
setting up 5-7

simple multilevel routing 8, 9
working 7, 8

routing modules **10**

S

scripts.test property **317**

secureShare function **267**

server environment
authbind, using 305
deploying to 301-304
multiple processes, using 306
screen, using 304

Server Name Indication. *See* **SNI**

session
site-wide session management 183, 184

site-wide session management
custom middleware 183, 184

SMS
automated phone call, making 247, 248
processed event listener, using 247
sending 242-245
sending, requirements 243
twilio.RestClient, working 246

SMTP server
about 256
creating 256-258
emails, receiving from 260, 261
working 259

SNI
about 265, 266
caveat 267

socket.io
about 128
callback function executing, over WebSockets 134-136
custom events 130, 131
fallback 128, 129
namespace 131-134
WebSockets, shared functions 136, 137
working 130

spawnInstance method **286**

SQL
about 85
command-line parsing module 89

connecting, to MySQL server 85-88
user input, cleaning 88
user input, using 88
square function method 137
startData event 259
static files
 favicon gotcha 15
 serving 10-14
 working 14
statStream method 289
stream.pipe method 20
streamStat method 292
super property 233

T

TCP
 about 249
 communicating with 250, 252
 port, forwarding 253
 traffic checking, pcap module used 254, 255
 working 252

test-driven module API
 creating 270-272
 unit testing, should.js used 274, 275
 working 273

test suite 270

throttle function 48

Transmission Control Protocol. *See TCP*

tweetPath method 80

Twilio dashboard 243

Twilio Numbers account section 244

TwiML 249

Twitter
 Google Hot Trends, cross referencing 77, 78, 79
 working with 74-77

U

url.parse method 9

V

validator module 235

views 203

virtual hosting paradigm

 about 261
 Express/Connect 264
 implementing 262
 SNI 265-267
 working 263

visitTag method 232, 233

W

WebSocket server

 creating 122-125
 node-based WebSocket client, creating 126, 127
 older Firefox browsers, supporting 126
 working 125

webSocketServer request event 126

whitelisting 27



Thank you for buying Node Cookbook

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

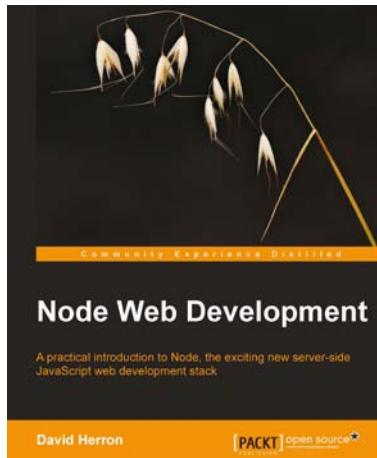
About Packt Open Source

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 Web Development

ISBN: 978-1-84951-514-6 Paperback: 172 pages

A practical introduction to Node, the exciting new server-side JavaScript web development stack

1. Go from nothing to a database-backed web application in no time at all
2. Get started quickly with Node and discover that JavaScript is not just for browsers anymore
3. An introduction to server-side JavaScript with Node, the Connect and Express frameworks, and using SQL or MongoDB database back-end



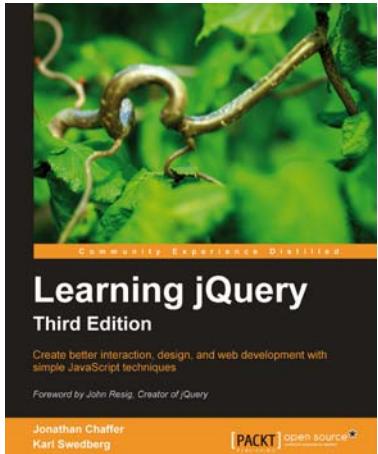
Learning Ext JS 4

ISBN: 978-1-84951-684-6 Paperback: 504 pages

Sencha Ext JS for a beginner

1. Learn the basics and create your first classes
2. Handle data and understand the way it works, create powerful widgets and new components
3. Dig into the new architecture defined by Sencha and work on real world projects

Please check www.PacktPub.com for information on our titles

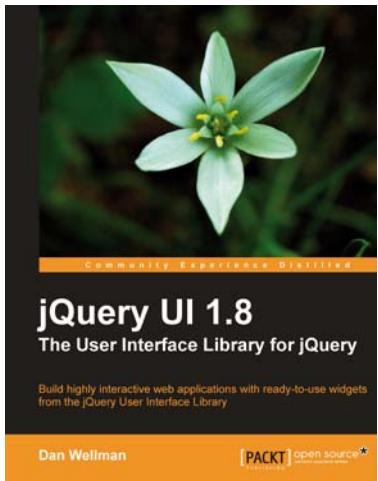


Learning jQuery, Third Edition

ISBN: 978-1-84951-654-9 Paperback: 428 pages

Create better interaction, design, and web development with simple JavaScript techniques

1. An introduction to jQuery that requires minimal programming experience
2. Detailed solutions to specific client-side problems
3. Revised and updated version of this popular jQuery book



jQuery UI 1.8: The User Interface Library for jQuery

ISBN: 978-1-84951-652-5 Paperback: 424 pages

Build highly interactive web applications with ready-to-use widgets from the jQuery User Interface Library

1. Packed with examples and clear explanations of how to easily design elegant and powerful front-end interfaces for your web applications
2. A section covering the widget factory including an in-depth example on how to build a custom jQuery UI widget
3. Updated code with significant changes and fixes to the previous edition

Please check www.PacktPub.com for information on our titles