Quick answers to common problems

Node Cookbook
Second Edition

Over 50 recipes to master the art of asynchronous server-side JavaScript using Node.js, with coverage of Express 4 and Socket.IO frameworks and the new Streams API

David Mark Clements

[PACKT] open source*
community experience distilled
PUBLISHING

Node Cookbook

Second Edition

Over 50 recipes to master the art of asynchronous server-side JavaScript using Node.js, with coverage of Express 4 and Socket.IO frameworks and the new Streams API

David Mark Clements

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Node Cookbook
Second Edition

First published: July 2012

Second edition: April 2014

Production Reference: 1180414

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

ISBN 978-1-78328-043-8

`www.packtpub.com`

Cover Image by Alvaro Dalloz (`alvaroff@gmail.com`)

Credits

Author
David Mark Clements

Reviewers
Vijay Annadi
Johannes Boyne
Aravind V.S

Commissioning Editor
Grant Mizen

Acquisition Editors
Antony Lowe
Sam Wood

Content Development Editor
Amey Varangaonkar

Technical Editors
Pratik More
Humera Shaikh
Ritika Singh

Copy Editors
Alisha Aranha
Mradula Hegde
Gladson Monteiro
Adithi Shetty

Project Coordinator
Amey Sawant

Proofreaders
Simran Bhogal
Maria Gould
Ameesha Green
Paul Hindle
Jonathan Todd

Indexer
Priya Subramani

Graphics
Sheetal Aute
Ronak Dhruv

Production Coordinator
Saiprasad Kadam

Cover Work
Saiprasad Kadam

About the Author

David Mark Clements is a JavaScript and Node specialist 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 9. David learned JavaScript at age 12, moving into Linux administration and PHP as a teenager.

Now (as a twenty something), he assists multinationals and startups alike with JavaScript solutions and training. Node has become a prominent member of his toolkit due to its versatility, vast ecosystem, and the cognitive ease that comes with full-stack JavaScript.

When he's not tinkering with computers, he's 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 the 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 gives me answers to problems when I'm not seeing the solution myself (Jms 1:5, 1 Cor 1:30).

About the Reviewers

Vijay Annadi is a freelance developer/architect with a passion for designing/developing complex yet simple software. Since 1997, he has been developing software applications using a wide array of languages and technologies, including Java, JavaScript, Python, Scala, and many others, with focus on both desktop and web applications.

Johannes Boyne is a full-stack developer, technical consultant, and entrepreneur. He co-founded Archkomm GmbH and is now working at Zweitag GmbH, a software engineering consultancy. His work with Node.js began with Version 0.4 and since then he has supported the Node.js community.

He started as a rich Internet application developer and did consulting work later on till he joined Archkomm for the VIRTUAL TWINS® project as technical lead. He is interested in new technologies such as NoSQL, high-performance and highly-scalable systems, as well as cloud computing. Besides work, he loves sports, reading about new scientific research, watching movies, and travelling.

He has also worked on books such as *Rich Internet Applications mit Adobe Flex 3*, *Simon Widjaja*, *Hanser Fachbuchverlag (2008)* and *Adobe Flex 4*, *Simon Widjaja*, *Hanser Fachbuchverlag (July 1, 2010)*. He was also a technical reviewer of the book *Node Security*, *Dominic Barnes*, *Packt Publishing*.

Aravind V.S is an aspiring mind and a creative brain to look forward to in the field of technology. He is a successful freelance software developer and web designer. His interest in embedded systems and computers paved his way into a programming career at the age of 15. He then developed an inventory management system for a local provision store and it rocketed his programming career sky high. His compassion and curiosity for technological advances and gadgets can be clearly seen on his blog `http://aravindvs.com/blog/`, where he talks about the current tech trends and also provides tutorials. He can be found outdoors focusing his camera or reading books during his leisure time.

I would like to take this opportunity to thank my friends and my parents for their support in completing the review of this book, especially my best friend Kavya Babu for her undying support and encouragement, without which I wouldn't be what I am today. Special thanks to Ryan Dahl and his team for NodeJS. Above all, I'd like to thank the Almighty for everything.

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.

PACKTLIB™

`http://PacktLib.PacktPub.com`

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

Why Subscribe?

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

Free Access for Packt account holders

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

Table of Contents

Preface

The principles of asynchronous event-driven programming are perfect for today's Web, where efficient, high-concurrency applications are essential for good user experience and a company's bottom line.

The use of Node for tooling and server-side logic with a browser-based client-side UI leads to a full-stack unilingual experience—everything is JavaScript. This saves developers, architects, project leads, and entire teams the cognitive energy of context-switching between languages, and yields rapid, fluid development cycles.

With a thriving community and success stories from major organizations (such as Groupon, PayPal, and Yahoo), Node.js is relevant to enthusiasts, start-ups, and enterprises alike.

Node Cookbook Second Edition shows you how to transfer your JavaScript skills to server-side programming. With simple examples and supporting code, this book 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.

The second edition comes with an additional chapter (*Chapter 5*, *Employing Streams*) and has been updated for the latest version of Node along with the most recent versions of the modules and frameworks discussed. In particular, the very latest versions of the popular **Express** and **Socket.IO** frameworks have extensive coverage.

Beginning with making your own web server, the practical recipes in this cookbook are designed to smoothly help you progress to make full web applications, command-line applications, and Node modules. *Node Cookbook Second Edition* 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 how to serve dynamic and static content, cache files in memory, stream large files straight from disk over HTTP, and secure your web server.

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

Chapter 3, *Working with Data Serialization*, explains how to convert data from your apps into XML and JSON formats when 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, *Employing Streams*, is included in the second edition. From streaming fundamentals to creating custom stream abstractions, this chapter introduces a powerful API that can boost the speed and memory efficiency of processing large amounts of data.

Chapter 6, *Going Real Time*, helps you to make real-time web apps with modern browser WebSocket technology, and gracefully degrade to long polling and other methods with Socket.IO.

Chapter 7, *Accelerating Development with Express*, explains how to leverage the Express framework to achieve rapid web development. It also covers the use of template languages and CSS engines, such as LESS and Stylus.

Chapter 8, *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 9, *Integrating Network Paradigms*, discusses how to send e-mails and create your own e-mail server, send SMS text messages, implement virtual hosting, and do fun and interesting things with raw TCP.

Chapter 10, *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 11, *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 workflow, or alternatively, simply use a Platform as a Service Provider.

What you need for this book

The following is a list of the software that is required to run the examples in this book:

- Windows, Mac OS X, or Linux
- Node 0.10.x or higher

The content and code will continue to be relevant for Node's 1.x.x releases.

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 Second Edition* 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 text styles that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can load a module into our app using Node's built-in `require` function."

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:

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
     /etc/asterisk/cdr_mysql.conf
```

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: "The console will say **foo doesn't exist**, because it doesn't."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

1

Making a Web Server

In this chapter, we will cover the following topics:

- 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 web server, customize it, and deliver content. All this can be done at the code level. This chapter demonstrates how to create a web server with Node and feed content through it, while implementing security and performance enhancements to cater for various situations.

> If we don't have Node installed yet, we can head to `http://nodejs.org` and hit the **INSTALL** button appearing on the homepage. This will download the relevant file to install Node on our operating system.

Setting up a router

In order to deliver web content, we need to make a **Uniform Resource Identifier** (**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 a general practice to call the `server.js` file (because the `npm start` command runs the `node server.js` command by default). We could put this new `server.js` file in a new folder.

It's also a good idea to install and use `supervisor`. We use `npm` (the module downloading and publishing command-line application that ships with Node) to install. On the command-line utility, we write the following command:

```
sudo npm -g install supervisor
```

> Essentially, `sudo` allows administrative privileges for Linux and Mac OS X systems. If we are using Node on Windows, we can drop the `sudo` part in any of our commands.

The `supervisor` module will conveniently autorestart our server when we save our changes. To kick things off, we can start our `server.js` file with the `supervisor` module by executing the following command:

```
supervisor server.js
```

> For more on possible arguments and the configuration of `supervisor`, check out `https://github.com/isaacs/node-supervisor`.

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 as follows:

```
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 e-mailed 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!** But 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 the `basename` variable of the path (the final part of the path) and reverse any URI encoding from the client with `decodeURI` as follows:

```
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 as follows:

```
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` error-related handling as follows:

```
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`, without which another page route would fail 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 filename 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, easier to extend, and can be seamlessly converted to JSON. We loop through our `pages` array using `forEach`.

Node is built on Google's V8 engine, which provides us with a number of **ECMAScript 5** (**ES5**) 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! The `forEach` function is an ES5 implementation; 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, and then we end the response with the object's output property.

The `response.end` method allows us to pass a parameter to it, which it writes just before finishing the response. In `response.end`, we have used a ternary operator (`?:`) to conditionally call `page.output` 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 `forEach` loop, `response.end` would never be called and 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.

The `response.finished` flag is affected by the `forEach` 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, the `forEach` loop does not operate asynchronously; it blocks until all loops are complete.

There's more...

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

Simple multilevel routing

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

```
var http=require('http');
var pages = [
  {route: '/', output: 'Woohoo!'},
```

```
  {route: '/about/this', output: 'Multilevel routing with Node'},
  {route: '/about/node', output: 'Evented I/O for V8 JavaScript.'},
  {route: '/another page', output: 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* recipe in this chapter.

Multilevel routing could be taken further; we could build and then traverse a more complex object as follows:

```
{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 alternatively create a helper function to define our routes that essentially pieces our object together for us. Alternatively, we could use one of the excellent noncore routing modules provided by the open source Node community. Excellent solutions already exist that provide helper methods to handle the increasing complexity of scalable multilevel routing. (See the *Routing modules* section and *Chapter 7*, *Accelerating Development with Express*).

Parsing the querystring module

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 the `url.parse` method is 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 as shown in the following code:

```
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`.

The routing modules

There's an up-to-date list of various routing modules for Node at `https://github.com/joyent/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.

> NodeZoo (`http://nodezoo.com`) is an excellent tool to research the state of a NODE module.

In *Chapter 7*, *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

- The *Serving static files* and *Securing against filesystem hacking exploits* recipes
- The *Implementing dynamic routing* recipe discussed in *Chapter 7*, *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 `http.createServer` callback. This is a basic conceptual starting point to serve 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`
- `styles.css`
- `script.js`

Add the following code to the HTML file `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>
```

Add the following code to the `script.js` JavaScript file:

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

And finally, add the following code to the CSS file `style.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 `fs` as well. With the help of the following code, let's create the server and use the `path` module to check if a file exists:

```
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';
  var 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 done it, then we can initialize our `server.js` file by running the following command:

```
supervisor server.js
```

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

```
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 (like `text/javascript`), without the server sending a `content-type` header, but 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 of 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 a `content-type` header by feeding `path.extname` into `mimeTypes` and then pass our retrieved `content-type` to `response.writeHead`. If the requested file doesn't exist, we'll write out a `404` error and end the response as follows:

```
//requires variables, mimeType object...
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();
    });
  }).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 as follows:

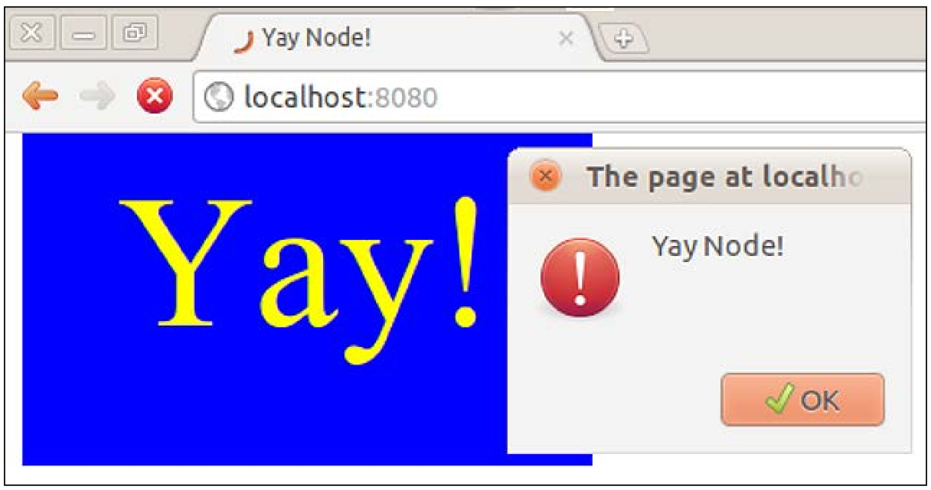
```
//http.createServer, inside fs.exists:
if (exists) {
  fs.readFile(f, function(err, data) {
    var headers={'Content-type': mimeTypes[path.extname(lookup)]};
    response.writeHead(200, headers);
    response.end(data);
  });
 return;
}
```

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

```
//requires variables, mimeType object...
//http.createServer,  path 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 of the `fs.readFile` callback. We are returning from the `fs.exists` callback to prevent further code execution (for example, sending the `404` error). Placing a `return` statement in an `if` statement is similar to using an `else` branch. However, the pattern of the `return` statement inside the `if` loop is encouraged instead of `if else`, as it eliminates a level of nesting. Nesting can be particularly prevalent in Node due to performing a lot of asynchronous tasks, which tend to use callback functions.

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. We can see the result 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 of serving 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 `||` (OR) 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 does exist, 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` as follows:

```
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
```

> `chmod` is a Unix-type system-specific command. If we are using Windows, there's no need to set file permissions in this case.

As long as we can access the file, we grab the `content-type` header 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` error message, and end the response.

There's more...

The favicon icon file is something to watch out for. We will explore the file in this section.

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 does begin to interfere, we can handle it as follows:

```
if (request.url === '/favicon.ico') {
  console.log('Not found: ' + f);
  response.end();
  return;
}
```

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

See also

- The *Caching content in memory for immediate delivery* recipe
- The *Optimizing performance with streaming* recipe
- The *Securing against filesystem hacking exploits* recipe

Caching content in memory for immediate delivery

Directly accessing storage on each client request is not ideal. For this task, we will explore how to enhance server efficiency by accessing the disk only on 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 following 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 respond to all requests for that file from memory afterwards. 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 and a new function called `cacheAndDeliver` have been added to store our files in memory. 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 as follows:

```
//...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 path exists code (404 handling)...
```

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

```
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 as we would just have 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, the 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 as follows:

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

That was easy! Now let's 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 (GNU's Not Unix!) coreutils `stat`. The `fs.stat` function 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 be updated but `mtime` would stay the same. We want to pay attention to permission changes as they happen so let's use the `ctime` property as shown in the following code:

```
//requires and mimeType object....
var cache = {};
function cacheAndDeliver(f, cb) {
  fs.stat(f, function (err, stats) {
    if (err) { return console.log('Oh no!, Eror', err); }
    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
  }); //end of fs.stat
}
```

[If we're using Node on Windows, we may have to substitute `ctime` with `mtime`, since `ctime` supports at least Version 0.10.12.]

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 dubbed `stats` using `Date.parse` to convert it to milliseconds since midnight, January 1st, 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, its 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 disk into the `cache` object.

See also

- The *Optimizing performance with streaming* recipe
- The *Browser-server transmission via AJAX* recipe in *Chapter 3*, *Working with Data Serialization*
- *Chapter 4*, *Interfacing with Databases*

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 a `response` object. 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 a 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.

> If streaming and piping are new concepts, don't worry! We'll be covering streams in depth in *Chapter 5*, *Employing Streams*.

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, which for the sake of simplicity would preferably be dealt with in the `http.createServer` callback. For brevity's sake, we will discard our `cacheAndDeliver` function and implement basic caching within the server callback as follows:

```
//...snip... requires, mime types, createServer, lookup and f
  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;
     } //...snip... rest of server code...
```

Later on, we will fill `cache[f].content` while we are interfacing with the `readStream` object. The following code shows how we use `fs.createReadStream`:

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

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

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

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

```
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 some 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 whole 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 as follows:

```
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!');
});
```

But what about ending the response? Conveniently, `stream.pipe` detects when the stream has ended and calls `response.end` for us. There's one other event we need to listen to, for caching purposes. 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;
    });
  }); //end of createReadStream
```

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.

> For this case, we're using the classic mode `data` event instead of the `readable` event coupled with `stream.read()` (see `http://nodejs.org/api/stream.html#stream_readable_read_size_1`) because it best suits our aim, which is to grab data from the stream as soon as possible. In *Chapter 5*, *Employing Streams*, we'll learn how to use the `stream.read` method.

How it works...

Instead of the client waiting for the server to load the entire file from 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 disk. The `fs.createReadStream` method creates a `readStream` object, which inherits from the `EventEmitter` class.

The `EventEmitter` class accomplishes the *evented* part of the *Node Cookbook Second Edition* tagline: *Evented I/O for V8 JavaScript*. Due to this, we'll be using listeners instead of callbacks to control the flow of stream logic.

We then added an `open` event listener using the `once` method since we want to stop listening to the `open` event once it is 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. If the client becomes overwhelmed with processing, `stream.pipe` applies *backpressure*, which means that the incoming stream is paused until the backlog of data is handled (we'll find out more about this in *Chapter 5*, *Employing Streams*).

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 class must be supplied with a size (or array or string), which in our case is the size of the file. To get the size, we used the asynchronous fs.stat method and captured the size property in the callback. The data event returns a Buffer variable as its only callback parameter.

The default value of bufferSize for a stream is 64 KB; any file whose size is less than the value of the bufferSize property will only trigger one data event because the whole file will fit into the first chunk of data. But for files that are greater than the value of the bufferSize property, 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 our readStream object 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 because 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 property 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, which tend to fragment a large amount of data, thus slowing down Node's ability to process them.

There's more...

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

Protecting against process memory overruns

Streaming allows for intelligent and minimal use of memory for processing large memory items. But even with well-written code, some apps may require significant memory.