

Node.js IN ACTION

Mike Cantelon
TJ Holowaychuk
Nathan Rajlich



MANNING



**MEAP Edition
Manning Early Access Program
Node.js in Action version 16**

Copyright 2013 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

brief contents

PART 1: NODE FUNDAMENTALS

Chapter 1: Welcome to Node.js

Chapter 2: Building a multi-room chat application

Chapter 3: Node programming fundamentals

PART 2: WEB APPLICATION DEVELOPMENT WITH NODE

Chapter 4: Building Node web applications

Chapter 5: Storing Node application data

Chapter 6: Testing Node applications

Chapter 7: Connect

Chapter 8: Connect's built-in middleware

Chapter 9: Express

Chapter 10: Web application templating

Chapter 11: Deploying Node web applications

PART 3: GOING FURTHER WITH NODE

Chapter 12: Beyond web servers

Chapter 13: The Node ecosystem

APPENDICES

Appendix A: Installing Node and community add-ons

Appendix B: Debugging Node

Welcome to Node.js

This chapter covers:

- What Node.js is
- JavaScript on the server
- Asynchronous and evented nature of Node
- Types of applications Node is designed for
- Sample Node programs

So what is Node.js? It's likely you have heard the term. Maybe you use Node. Maybe you are curious about it. At this point in time, Node is very popular and young (it debuted in 2009¹). It is the second most watched project on GitHub², has quite a following in its Google group³ and IRC channel⁴ and has more than 15,000 community modules published in NPM (the package manager)⁵. All this to say, *there is considerable traction behind this platform.*

Footnote 1 First talk on Node by creator Ryan Dahl - http://jsconf.eu/2009/video_nodejs_by_ryan_dahl.html

Footnote 2 <https://github.com/popular/starred>

Footnote 3 <http://groups.google.com/group/nodejs>

Footnote 4 <http://webchat.freenode.net/?channels=node.js>

Footnote 5 <http://npmjs.org>

The official website (nodejs.org) defines Node as "a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js

uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.”

In this chapter, we'll look at:

- Why JavaScript matters for server side development
- How the browser handles I/O using JavaScript
- How Node handles I/O on the server
- What are DIRTY applications and why they are a good fit for Node
- A sampling of a few basic Node programs

Let's first turn our attention to JavaScript...

1.1 Built on JavaScript

For better or worse JavaScript is the world’s most popular programming language⁶. If you have done any programming for the web, it is unavoidable. JavaScript, because of the sheer reach of the web, is the “*write once, run anywhere*” dream that Java had back in the 1990s.

Footnote 6 “JavaScript: Your New Overlord” - http://www.youtube.com/watch?v=Trurfqh_6fQ

Around the “Ajax revolution” in 2005, JavaScript went from being a “toy” language to something people write real and significant programs with. Some of the notable firsts were Google Maps and Gmail but today there are a host of web applications from Twitter to Facebook to Github.

Since the release of Google Chrome in late 2008, JavaScript performance has been improving at an incredibly fast rate due to heavy competition between browser vendors (i.e. Mozilla, Microsoft, Apple, Opera, and Google). The performance of these modern JavaScript virtual machines are literally changing the types of applications we can build on the web⁷. A compelling and frankly mind-blowing example of this is jslinux⁸, a PC emulator running in JavaScript where you can load a Linux kernel, interact with the terminal session, and compile a C program all in your browser.

Footnote 7 For some examples: <http://www.chromeexperiments.com/>

Footnote 8 <http://bellard.org/jslinux/>

Node specifically uses V8, the virtual machine that powers Google Chrome, for server-side programming. V8 gives a huge boost in performance because it cuts out

the middleman, preferring straight compilation into native machine code over executing bytecode or using an interpreter. Because Node uses JavaScript on the server there are other benefits:

- Developers can write web applications in one language, which helps by: reducing the "context" switch between client and server development, and allowing for code sharing between client and server (e.g. reusing the same code for form validation or game logic).
- JSON is a very popular data interchange format today and it is native JavaScript.
- JavaScript is the language used in various NoSQL databases (e.g. CouchDB/MongoDB) so interfacing with them is a natural fit (e.g. MongoDB shell and query language is JS, CouchDB map/reduce is JS).
- JavaScript is a compilation target and there are a number of languages that compile to it already⁹.

F o o t n o t e

9

<https://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS>

- Node uses one virtual machine (V8) that keeps up with the ECMAScript¹⁰ standard. In other words, you don't have to wait for all the browsers to catch up to use new JavaScript language features in Node.

Footnote 10 <http://en.wikipedia.org/wiki/ECMAScript>

Who knew JavaScript would end up being a compelling language for writing server-side applications? Yet, due to its sheer reach, performance, and other characteristics mentioned previously, Node has gained a lot of traction. JavaScript is only one piece of puzzle though, the *way* Node uses JavaScript is even more compelling. To understand the Node environment, let's dive into the JavaScript environment we are most familiar with: the browser.

1.2 Asynchronous and Evented: The Browser

Node provides an event-driven and asynchronous platform for server-side JavaScript. It brings JavaScript to the server much in the same way a browser brings JavaScript to the client. It is important to understand how the browser works in order to understand how Node works. Both are event-driven (i.e. use an event loop¹¹) and non-blocking when handling I/O (i.e. use asynchronous I/O¹²). Let's look an example to explain what that means.

Footnote 11 http://en.wikipedia.org/wiki/Event_loop

Footnote 12 http://en.wikipedia.org/wiki/Asynchronous_I/O

Take this common snippet of jQuery performing an Ajax request using XHR¹³:

Footnote 13 <http://en.wikipedia.org/wiki/XMLHttpRequest>

```
$.post('/resource.json', function (data) {
  console.log(data);
});
// script execution continues
```

➊ I/O does not block execution

In this program we perform an HTTP request for `resource.json`. When the response comes back, an anonymous function is called (a.k.a. the "callback" in this context) containing the argument `data`, which is the data received from that request.

Notice that the code was *not* written like this:

```
var data = $.post('/resource.json');
console.log(data);
```

➊ I/O blocks execution until finished

In this example, the assumption is that the response for `resource.json` would be stored in the `data` variable *when it is ready* and that the `console.log` function *will not execute until then*. The I/O operation (e.g. Ajax request) would "block" script execution from continuing until ready. Since the browser is *single-threaded*, if this request took 400ms to return, any other events happening on that page would wait until then before execution. You can imagine the poor user experience, for example, if an animation was paused or the user was trying to interact with the page somehow.

Thankfully that is not the case. When I/O happens in the browser, it happens outside of the event loop (i.e. main script execution) and then an "event" is emitted when the I/O is finished¹⁴ which is handled by a function (often called the "callback") as seen in figure 1.1:

Footnote 14 For completeness, there are a few exceptions which "block" execution in the browser and usage is typically discouraged: alert, prompt, confirm, and synchronous XHR.

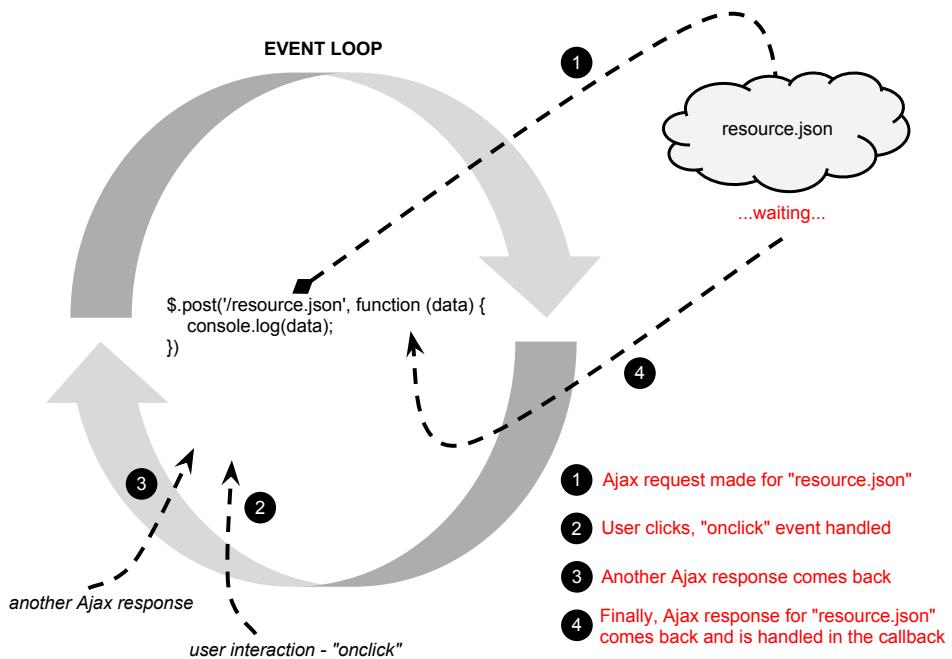


Figure 1.1 An example of non-blocking I/O in the browser

The I/O is happening asynchronously and not "blocking" the script execution allowing the event loop to respond to whatever other interactions or requests that are being performed on the page. This enables the browser to be responsive to the client and handle a lot of interactivity on the page.

Make a note of that and let's switch over to the server.

1.3 Asynchronous and Evented: The Server

For the most part, we are familiar with a conventional I/O model for server-side programming like the "blocking" jQuery example in section 1.2. Here's an example of how it looks in PHP:

```
$result = mysql_query('SELECT * FROM myTable');
print_r($result);
```

1 Execution stops until DB query completes

We are doing some I/O and the process is blocked from continuing until all the data has come back. For many applications this model is fine and is easy to follow. What may not be apparent is that the process has state/memory and is essentially doing "nothing" until the I/O is completed. That could take 10ms to minutes depending on the latency of the I/O operation. Latency can be unexpected as well, for example:

- The disk is performing a maintenance operation, pausing reads/writes
- A database query is slower because of increased load
- Pulling a resource from "sitexyz.com" is sluggish today for some reason

If a program blocks on I/O, what does the server do when there are more requests to handle? Typically this means that we use a multi-threaded approach. A common implementation is to use one thread per connection and setup a thread pool for those connections. You can think of threads as computational workspaces in which the processor works on one task. In many cases, a thread is contained inside a process and maintains its own working memory. Each thread handles one or more server connections. While this sounds like a natural way to delegate server labor, to developers who've been doing this a long time, managing threads within an application can be complex. Also, when a large number of threads is needed to handle many concurrent server connections, threading can tax operating system resources. Threads require CPU to perform context-switches as well as additional RAM.

To illustrate this, let's look at a benchmark (shown in Figure 1.2) comparing NGINX and Apache. NGINX¹⁵, if you aren't familiar with it, is an HTTP server like Apache but instead of using the multi-threaded approach with blocking I/O, it uses an event loop with asynchronous I/O (like the browser and Node). Because of these design choices, NGINX is often able to handle more requests and connected clients, making it a more responsive solution.¹⁶

Footnote 15 <http://nginx.com/>

Footnote 16 If you are more interested in this problem: <http://www.kegel.com/c10k.html>

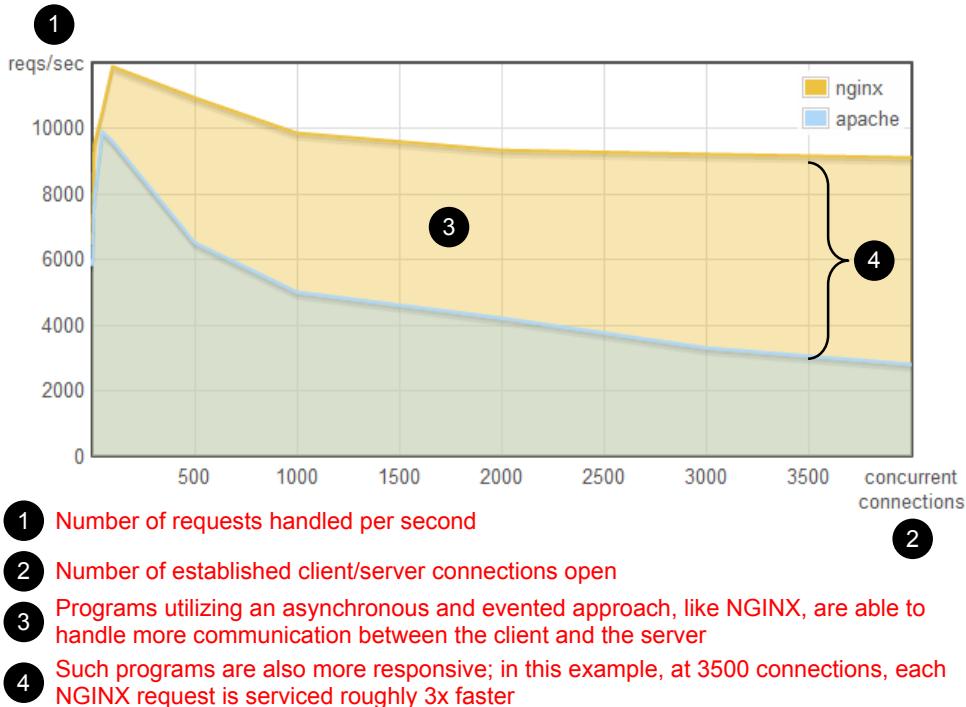


Figure 1.2 WebFaction Apache / NGINX benchmark:
<http://jppommet.com/from-webfaction-a-little-holiday-present-1000>

In Node, I/O almost always is performed outside of the main event loop allowing the server to stay efficient and responsive like NGINX. This makes it much harder for a process to become I/O bound because I/O latency isn't going to crash your server or use the resources it would if you were blocking. It allows the server to be lightweight on what are typically the slowest operations a server performs.¹⁷.

Footnote 17 More details at <http://nodejs.org/about/>

This mix of an event-driven and asynchronous model and the widely accessible JavaScript language helps open up a exciting world of data-intensive real-time applications.

1.4 DIRTy Applications

There actually is acronym for the types of applications Node is designed for: *DIRT*. It stands for *data-intensive real-time* applications. Since Node itself is very lightweight on I/O, it is good at shuffling/proxying data from one pipe to another. It allows a server to hold open a number of connections while handling many requests and keeping a small memory footprint. It is designed to be responsive like the browser.

Real-time applications are a new use-case of the web. Many web applications

now provide information virtually instantly, implementing things like: online whiteboard collaboration, realtime pinpointing of approaching public transit buses, and multiplayer games. Whether its existing applications being enhanced with real-time components or completely new types of applications, the web is moving towards more responsive and collaborative environments. These new types of web applications, however, call for a platform that can respond almost instantly to a large number of concurrent users. Node is good at this and not just for web, but also other I/O heavy applications.

A good example of a DIRTY application written with Node is Browserling¹⁸ (shown in Figure 1.3). The site allows in-browser use of other browsers. This is extremely useful to front-end web developers as it frees them from having to install numerous browsers and operating systems solely for testing. Browserling leverages a Node-driven project called StackVM, which manages virtual machines (VMs), created using the QEMU ("Quick Emulator") emulator. QEMU emulates the CPU and peripherals needed to run the browser.

Footnote 18 browserling.com

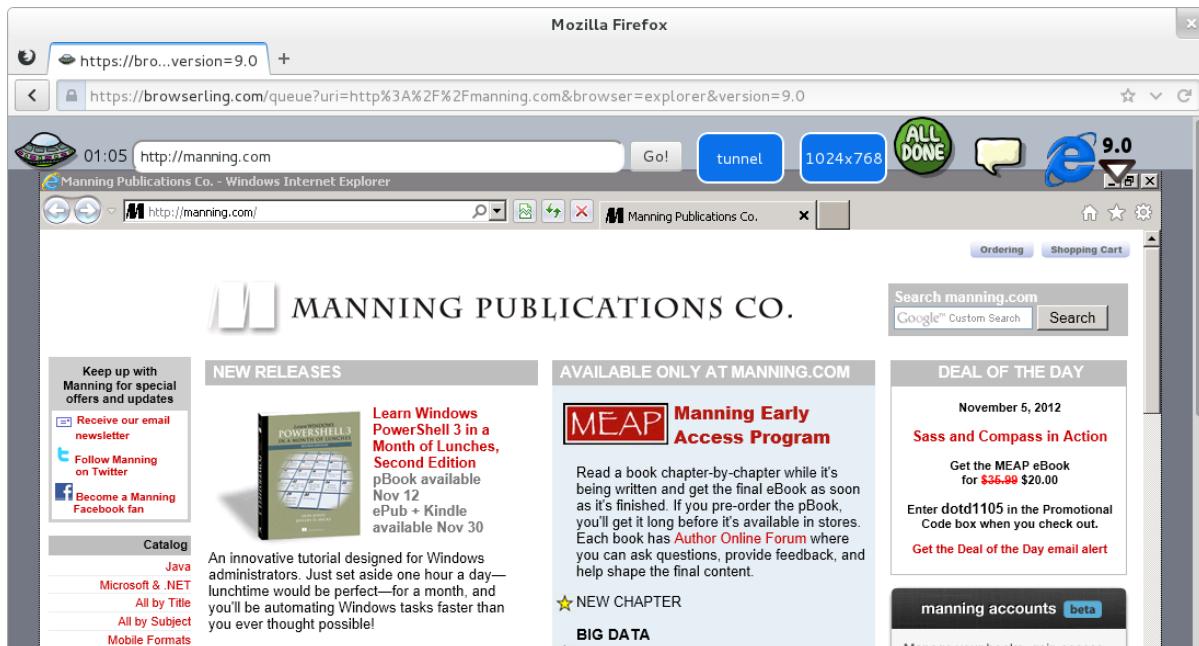
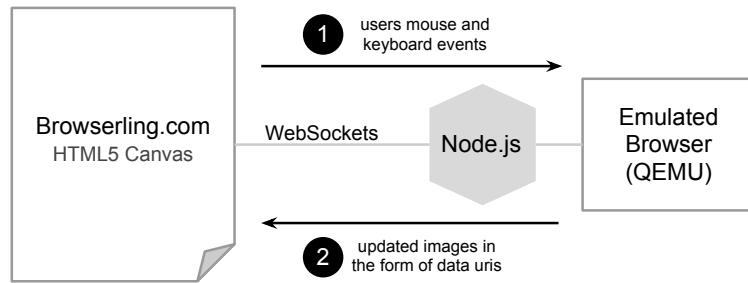


Figure 1.3 Browserling: Interactive cross-browser testing utilizing Node.js

Browserling has VMs run test browsers and then relays the keyboard and mouse input data from the user's browser to the emulated browser which, in turn, streams the repainted regions of the emulated browser and redraws them on the canvas of the user's browser. This is illustrated in figure 1.4.



- 1 In the browser, user's mouse and keyboard events are passed over WebSockets in real time to Node.js which in turn passes those to the emulator
- 2 The repainted regions of the emulated browser affected by the user interaction are streamed back through Node and WebSockets and drawn on the canvas in the browser

Figure 1.4 Browserling Workflow

Browserling also provides a complimentary project using Node called Testling¹⁹ allowing you to run test suites against multiple browsers in parallel from the command line.

Footnote 19 testling.com

Browserling and Testling are good examples of a DIRTY applications and the infrastructure for building a scalable network applications like it are at play when you sit down to write your first Node application. Let's take a look at how Node's API provides this tooling right out of the box.

1.5 DIRTY by Default

Node was built from the ground up to have an event-driven and asynchronous model. JavaScript has never had standard I/O libraries, which are common to server-side languages, the "host" environment has always determined this for JavaScript. The most common "host" environment for JavaScript, the one most developers are used to, is the browser which is event-driven and asynchronous.

Node tries to keep consistency between the browser by reimplementing common "host" objects, for example:

- Timer API (e.g. `setTimeout`)

- Console API (e.g. `console.log`)

Node also includes a "core" set of modules for many types of network and file I/O. These include modules for: HTTP, TLS, HTTPS, File System (POSIX), Datagram (UDP), and NET (TCP). The "core" is intentionally small, low-level, and uncomplicated including just the building blocks for I/O based applications. 3rd-party modules build upon these blocks to offer greater abstractions for common problems.

SIDE BAR Platform vs. Framework

Node is a platform for JavaScript applications which is not to be confused with a framework. It is a common misconception to think of Node as "Rails" or "Django" for JavaScript when it is much lower level. Although if you are interested in frameworks for web applications, we will talk about a popular one for Node called Express later on in this book.

After all this discussion you probably are wondering what does Node code look like? Let's cover a few simple examples:

- A simple asynchronous example
- A "hello world" web server
- An example of streams

1.5.1 Simple Async Example

In section 1.2, we looked at this Ajax example using jQuery:

```
$.post('/resource.json', function (data) {
  console.log(data);
});
```

Let's do something similar in Node but instead using the file system (`fs`) module to load `resource.json` from disk. Notice how similar the program is compared to the previous jQuery example.

```
var fs = require('fs');
fs.readFile('./resource.json', function (er, data) {
  console.log(data);
```

```
})
```

In this program, we read the `resource.json` file from disk. When all the data is read, an anonymous function is called (a.k.a. the "callback") containing the arguments `err`, if any error occurred, and `data`, which is the file data.

The process "loops" behind the scenes able to handle any other operations that may come its way until the data is ready. All the evented/async benefits we talked about earlier are in play automatically. The difference here is that instead of making an Ajax request from the browser using jQuery, we are accessing the file system in Node to grab `resource.json`. This latter action is illustrated in figure 1.5.

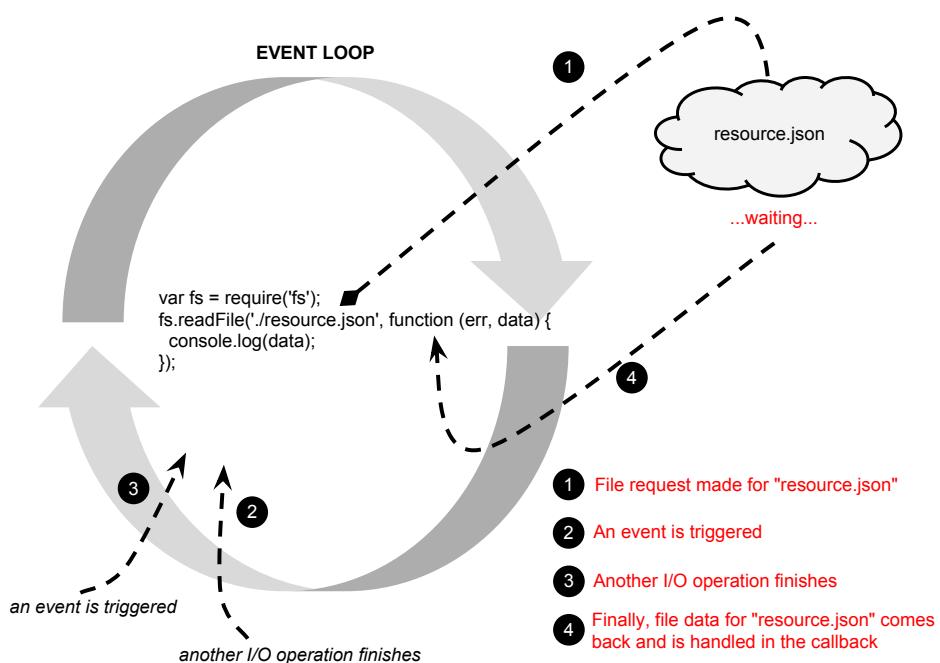


Figure 1.5 An example of non-blocking I/O in Node

1.5.2 Hello World HTTP Server

A very common use case for Node is building servers. Node makes it very simple to create different types of servers. This can feel odd if you are used to having a server 'host' your application (e.g. a PHP application hosted on Apache HTTP server). In Node, the server and the application *are the same*. Here is an example of an HTTP server that simply responds to any request with 'Hello World':

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(3000);
console.log('Server running at http://localhost:3000/');
```

Whenever a request happens, the function (`req, res`) "callback" is fired and "Hello World" is written out as the response. This event model is akin to listening to an `onclick` event in the browser. A 'click' could happen at any point so you setup a function to perform some logic whenever that happens. Here, Node provides a function for a 'request' whenever that happens. Another way to write this same server to make the 'request' event even more explicit is:

```
var http = require('http');
var server = http.createServer();
server.on('request', function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
})
server.listen(3000);
console.log('Server running at http://localhost:3000/');
```

1 Setting up an event listener for 'request'

1.5.3 Streaming Data

Node is also huge on streams and streaming. You can think of streams like arrays but instead of having data distributed over space, streams can be thought of as *data distributed over time*. By bringing data in chunk by chunk, the developer is given the ability to handle that data as it comes in instead of waiting for it all to arrive before acting. Here's how we would stream `resource.json`:

```
var stream = fs.createReadStream('./resource.json')
stream.on('data', function (chunk) {
  console.log(chunk)
})
stream.on('end', function () {
  console.log('finished')
})
```

1 'data' event fires when new chunk is ready

data events are fired whenever a new chunk of data is ready and end is fired

when all the chunks have been loaded. A 'chunk' can vary in size depending on the type of data. This low level access to the read stream allows you efficiently deal with data as it is read instead of waiting for it all to buffer in memory.

Node also provides writable streams that you can write chunks of data to. One of those is the response (`res`) object when a 'request' happens on an HTTP server.

Readable and writeable streams can be connected to make pipes much like the `| (pipe)` operator in shell scripting. This provides an efficient way to write out data as soon as it's ready without waiting for the complete resource to be read and then written out. Let's use our HTTP server from before to illustrate streaming an image to a client:

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'image/png'});
  fs.createReadStream('./image.png').pipe(res);
}).listen(3000);
console.log('Server running at http://localhost:3000/');
```

1 Piping from a readable stream to a writeable stream.

In this one-liner, the data is read in from the file (`fs.createReadStream`) and being sent out (`.pipe`) to the client (`res`) as it comes in. The event loop is able to handle other events while data is being streamed.

Node provides this "DIRTY by default" approach across multiple platforms including various UNIXes and Windows. The underlying asynchronous I/O library (libuv) was built specifically to provide a unified experience regardless of the parent operating system which allows programs to be more easily ported across devices and run on multiple devices if needed.

1.6 In Summary

Like any technology, Node is not a silver bullet, it just tries to tackle certain problems and open new possibilities. One of the interesting things about Node is it brings people from all aspects of the system together. Many come to Node being JavaScript client-side programmers, others are server-side programmers, and others are systems level programmers. Wherever you fit, we hope you have an understanding of where Node may fit in your stack. To review:

Node is:

- Built on JavaScript

- Evented and asynchronous
- For data-intensive real-time applications

In Chapter 2, we will build a simple DIRTy web application so you can see how a Node application works.



Building a multi-room chat application

This chapter covers:

- A first look at various Node components
- A sample real-time application using Node
- Server and client-side interaction

In chapter 1, you learned how asynchronous development using Node differs from conventional/synchronous development. In this chapter, we'll take a practical look at Node by creating a small event-driven chat application. Don't worry if the details in this chapter seem over your head: the intent is to demystify Node development and give you a preview of what you'll be able to do once you've completed the book.

This chapter assumes you: have experience with web application development, have a basic understanding of HTTP, and are familiar with jQuery. As you move through the chapter, you'll:

- Tour the application to see how it will work
- Review technology requirements and perform the initial application setup
- Serve the application's HTML, CSS, and client-side JavaScript
- Handle chat-related messaging using Socket.io
- Use client-side JavaScript for the application's UI

Let's start with an application overview--what it will look like and how it will behave when it's completed.

2.1 Application overview

The application allows users to chat online with each other by entering messages into a simple form as shown in figure 2.1. A message, once entered, is sent to all other users in the same chat room.

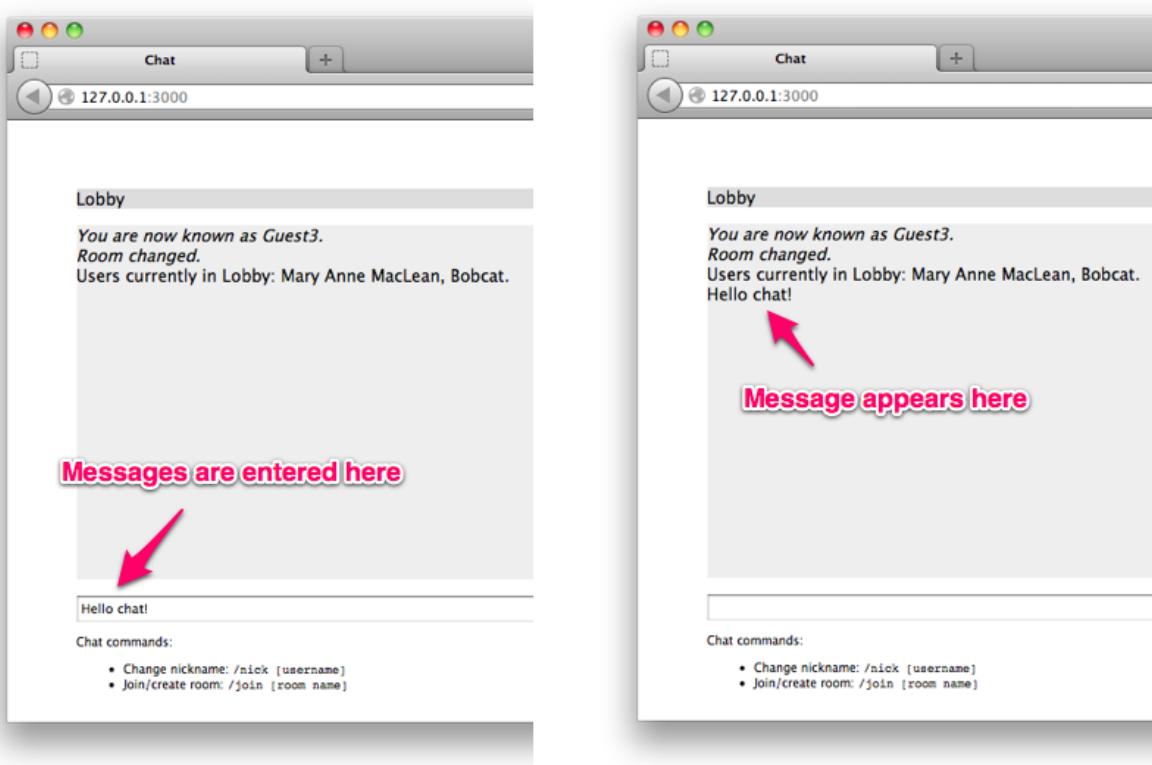


Figure 2.1 Entering a message into chat

When starting the application, a user is automatically assigned a guest name, but can change it by entering a command, as shown in figure 2.2. Chat commands are prefaced with a "/".

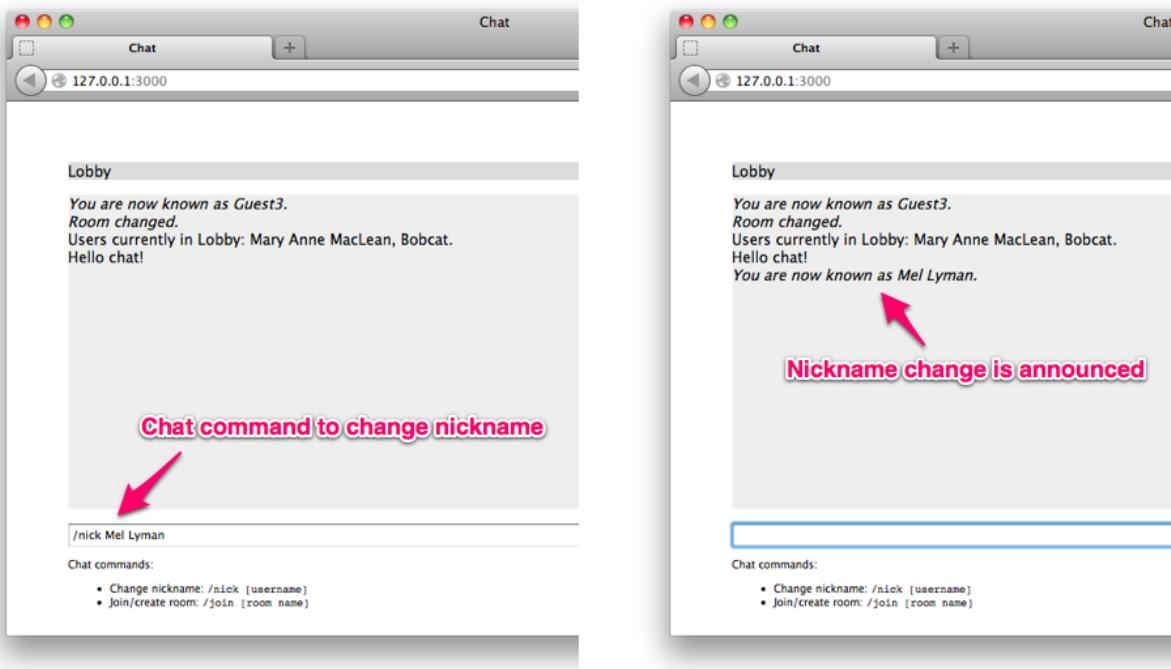


Figure 2.2 Changing one's chat name

Similarly, a user can enter a command to create a new room (or change to it if it already exists), as shown in figure 2.3. When joining or creating a room, the new room name will be shown in the horizontal bar at the top of the chat application. The room will also be included in the list of available rooms to the right of the chat message area.

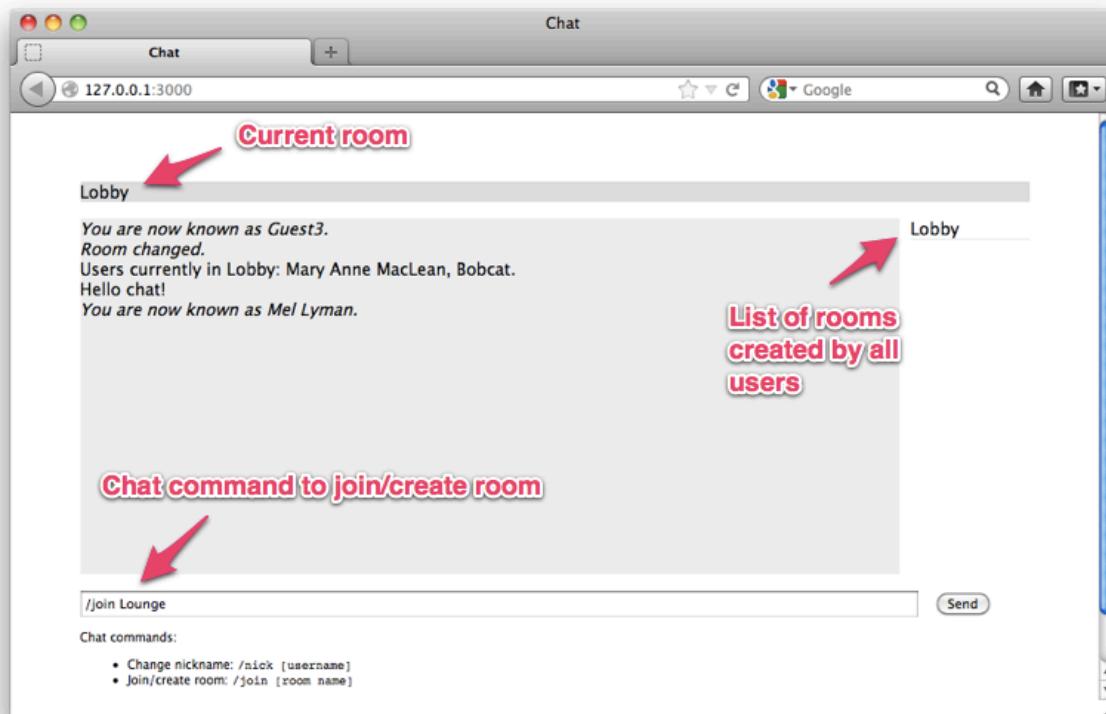


Figure 2.3 Changing rooms

After changing to a new room the system will confirm it, as shown in figure 2.4.

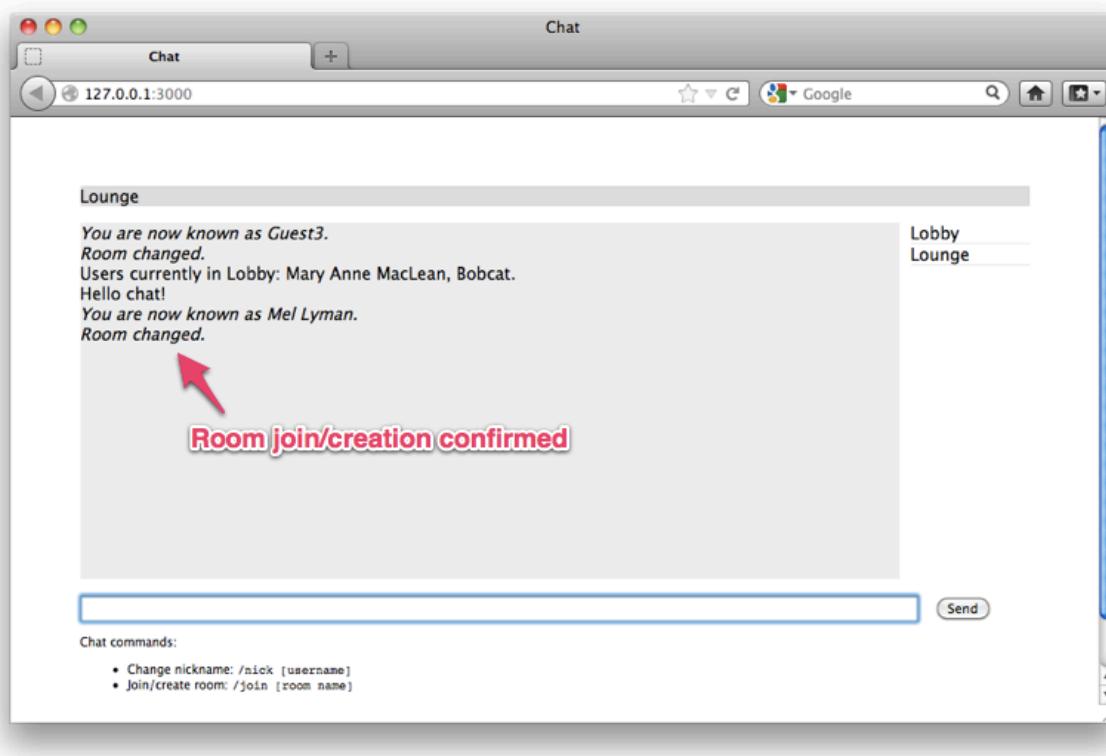


Figure 2.4 The results of changing to a new room

While the functionality of this application is deliberately barebones, it showcases important components and fundamental techniques needed to create a real-time web application. The application shows how Node can simultaneously serve conventional HTTP data (like static files) and real-time data (chat messages). It also shows how Node applications are organized and how dependencies are managed.

Let's now look at the technologies needed to implement this application.

2.2 Application requirements and initial setup

The chat application you'll create needs to:

- serve static files (such as HTML, CSS, and client-side JavaScript),
- handle chat-related messaging on the server, and
- handle chat-related messaging in the user's web browser

To serve static files, we'll use Node's built-in "http" module. However, when serving files via HTTP, it's usually not enough to just send the contents of a file; you also should include the type of file being sent. This is done by setting the "Content-Type" HTTP header with the proper MIME¹ type for the file. To look up these MIME types we will use a 3rd-party module called "mime".

Footnote 1 <http://en.wikipedia.org/wiki/MIME>

To handle chat-related messaging, we could poll the server with Ajax. However, to make this application as responsive as possible, it will avoid using traditional Ajax means to send messages. Ajax uses HTTP as a transport mechanism and HTTP wasn't designed for real-time communication. When a message is sent using HTTP, a new TCP/IP connection must be used. Opening and closing connections takes time and the size of the data transfer is larger since HTTP headers are sent on every request. Instead of employing a solution reliant on HTTP, the application will prefer WebSocket², which was designed as a bidirectional lightweight communications protocol to support real-time communication.

Footnote 2 <http://en.wikipedia.org/wiki/WebSocket>

As only HTML5 compliant browsers, for the most part, support WebSocket, the application will leverage the popular Socket.io³ library which provides a number of fallbacks, including the use of Flash, should using WebSocket not be possible.

Socket.io handles fallback functionality transparently, requiring no additional code or configuration. Socket.io is covered more deeply in chapter 12: Beyond Web Servers.

Footnote 3 <http://socket.io/>

Before we get in and actually do the preliminary work of setting up the application's file structure and dependencies, let's talk more about how Node lets you simultaneously handle HTTP and WebSocket – one of the reasons why it's so good for real-timey applications.

2.2.1 Serving HTTP and WebSocket

Although we're avoiding the use of Ajax for sending and receiving chat messages, the application will still use HTTP to deliver the necessary HTML, CSS, and client-side JavaScript needed to set things up in the user's browser.

Node can handle simultaneously serving HTTP and WebSocket easily, using a single TCP/IP port, as figure 2.5 shows visually. Node comes with a module that provides HTTP serving functionality. Then are a number of third-party Node modules, such as Express, which build upon Node's built-in functionality to make web serving even easier. We'll go into depth about how to use Express to build web application in chapter 9. In this chapter's application, however, we'll stick to the basics.

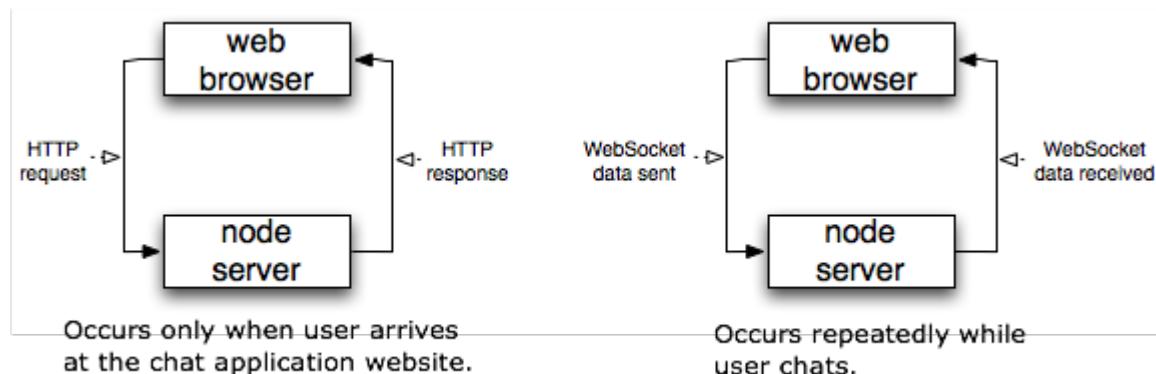


Figure 2.5 Handling HTTP and WebSocket within a single application

Now that you have a rough idea of the core technologies the application will use, let's start fleshing it out.

SIDE BAR
Need to install Node?

If you haven't already installed Node, please head to appendix A now for instructions for doing so.

2.2.2 Creating the application file structure

To start constructing the tutorial application, create a project directory for it. The main application file will go directly in this directory. You'll need to add a "lib" subdirectory, within which some server-side logic will be placed. You'll need to create a "public" subdirectory where client-side files will be placed. Within the "public" subdirectory create a "javascripts" subdirectory and a "stylesheets" directory.

Your directory structure should now look like figure 2.6. Note that while we've chosen to organize the files in a particular way in this chapter, Node doesn't require you to maintain any particular directory structure: application files can be organized in any way that makes sense to you.

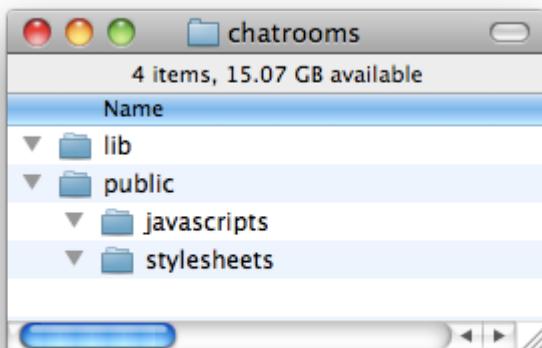


Figure 2.6 The skeletal project directory for the chat application.

Now that we've established a directory structure, you'll want to specify the application's dependencies.

An application dependency, in this context, is a module that needs to be installed to provide functionality needed by the application. Let's say, for example, that you were creating an application that needed to access data stored using a MySQL database. Node doesn't come with a built-in module that allows access to MySQL so you'd have to install a 3rd-party one and this would be considered a dependency.

2.2.3 Specifying dependencies

Although you can create Node applications without formally specifying dependencies, it's a good habit to take the time to specify them. This way if you want others to use your application, or you plan on running it in more than one place, it becomes more straightforward to set up.

Application dependencies are specified using a `package.json` file. This file is always placed in an application's root directory. A `package.json` file consists of a JSON expression that follows the CommonJS package descriptor standard⁴ and describes your application. In a `package.json` file you can specify many things, but the most important are the name of your application, the version, a description of what the application does, and the application's dependencies.

Footnote 4 <http://wiki.commonjs.org/wiki/Packages/1.0>

Listing 2.1 contains a package descriptor file that describes the functionality and dependencies of the tutorial application. Save this file as "package.json" in the root directory of the tutorial application.

Listing 2.1 package.json: A package descriptor file

```
{
  "name": "chatrooms", ①
  "version": "0.0.1",
  "description": "Minimalist multi-room chat server",
  "dependencies": { ②
    "socket.io": "~0.9.6",
    "mime": "~1.2.7"
  }
}
```

- ① Name of package
- ② Package dependencies

If the content of this file seems a bit confusing, don't worry... you'll learn about `package.json` files in depth in chapter 13: The Node Ecosystem.

2.2.4 Installing dependencies

With a package.json file defined, installing your application's dependencies becomes trivial. The Node Package Manager (npm)⁵ is a utility that comes bundled with Node. It offers a great deal of functionality, allowing you to easily install third-party Node modules and globally publish any Node modules you yourself create. Another thing it can do is read dependencies from package.json files and install each of them with a single command.

Footnote 5 <https://github.com/isaacs/npm>

Enter the following command in the root of your tutorial directory.

```
npm install
```

If you look in the tutorial directory now there should be a newly created node_modules directory, as shown in figure 2.7. This directory contains your application's dependencies.

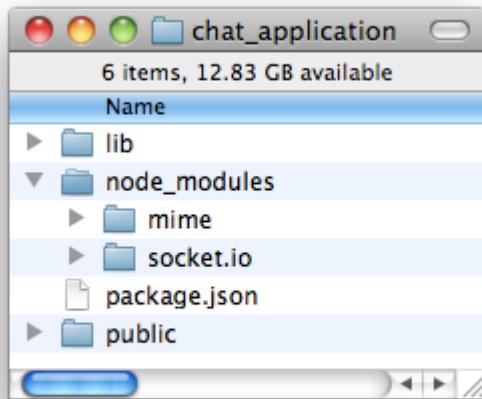


Figure 2.7 When using npm to install dependencies a "node_modules" directory is created.

With the directory structure established and dependencies installed, you're ready to start fleshing out the application logic.

2.3 Serving the application's HTML, CSS, and client-side JavaScript

As outlined earlier, the chat application needs to be capable of three basic things:

- serving static files to the user's web browser
- handling chat-related messaging on the server
- handling chat-related messaging in the user's web browser

Application logic will be handled by a number of files, some run on the server and some run on the client, as shown in figure 2.8. The JavaScript files run on the client need to be served as static assets, rather than executed by Node.

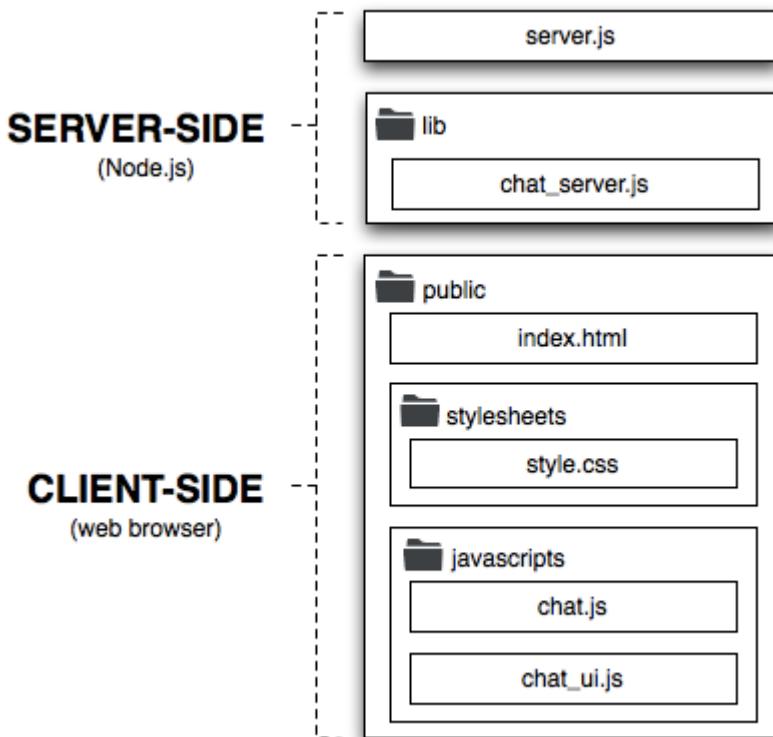


Figure 2.8 When using npm to install dependencies a "node_modules" directory is created.

In this section, tackling the first of those requirements, we'll define the logic needed to serve static files. We'll then add the static HTML and CSS files themselves.

2.3.1 Creating a basic static file server

To create a static file server, we'll leverage some of Node's built-in functionality as well as the third-party "mime" add-on for determining a file MIME type.

To start the main application file, create a file named `server.js` in the root of your project directory and put the listing 2.2's variable declarations in it. These

declarations will give you access to Node's HTTP-related functionality, the ability to interact with the filesystem, functionality related to file paths, and the ability to determine a file's MIME type. The `cache` variable will be used to cache file data.

Listing 2.2 server.js: variable declarations

```
var http = require('http');      ①
var fs = require('fs');          ②
var path = require('path');       ③
var mime = require('mime');      ④
var cache = {};                 ⑤
```

- ① The built-in "http" module provides HTTP server and client functionality
- ② The built-in "fs" module provides filesystem-related functionality
- ③ The built-in "path" module provides filesystem path-related functionality
- ④ The add-on "mime" module provides the ability to derive a MIME type based on a filename extension
- ⑤ The cache object is where the contents of cached files are stored

SENDING FILE DATA AND ERROR RESPONSES

Next you'll add three helper functions used for serving static HTTP files. The first will handle the sending of 404 errors for when a file is requested that doesn't exist. Add the following helper function to `server.js`.

```
function send404(response) {
  response.writeHead(404, {'Content-Type': 'text/plain'});
  response.write('Error 404: resource not found.');
  response.end();
}
```

The second helper function handles serving file data. The function first writes the appropriate HTTP headers then sends the contents of the file. Add the following code to `server.js`.

```
function sendFile(response, filePath, fileContents) {
  response.writeHead(
    200,
    {"content-type": mime.lookup(path.basename(filePath))})
  );
```

```

    response.end(fileContents);
}

```

Accessing memory storage (RAM) is faster than accessing the filesystem. Because of this, it's common for Node applications to cache frequently used data in memory. Our chat application will cache static files to memory, only reading them from disk the first time they are accessed.

The next helper, detailed in listing 2.3, determines whether or not a file is cached and, if so, serves it. If a file isn't cached, it is read from disk and served. If the file doesn't exist, an HTTP 404 error is returned as a response. Add this helper function, as well, to `server.js`.

Listing 2.3 server.js: Serving static files

```

function serveStatic(response, cache, absPath) {
  if (cache[absPath]) { ①
    sendFile(response, absPath, cache[absPath]); ②
  } else {
    fs.exists(absPath, function(exists) { ③
      if (exists) {
        fs.readFile(absPath, function(err, data) { ④
          if (err) {
            send404(response);
          } else {
            cache[absPath] = data;
            sendFile(response, absPath, data); ⑤
          }
        });
      } else {
        send404(response); ⑥
      }
    });
  }
}

```

- ① Check if file is cached in memory
- ② Serve file from memory
- ③ Check if file exists
- ④ Read file from disk
- ⑤ Serve file read from disk
- ⑥ Send HTTP 404 response

CREATING THE HTTP SERVER

For the HTTP server, an anonymous function is provided as an argument to `createServer`, acting as a callback that defines how each HTTP request should be handled. The callback function accepts two arguments: `request` and `response`. When the callback executes, the HTTP server will populate the `request` and `response` arguments with objects that, respectively, allow you to work out the details of the request and send back a response. You'll learn about Node's HTTP module in detail in chapter 4, Building Node web applications. Now, let's add the logic in listing 2.4 to create the HTTP server.

Listing 2.4 server.js: Logic to create an HTTP server

```

var server = http.createServer(function(request, response) { ①
  var filePath = false;

  if (request.url == '/') {
    filePath = 'public/index.html'; ②
  } else {
    filePath = 'public' + request.url; ③
  }

  var absPath = './' + filePath;
  serveStatic(response, cache, absPath); ④
});
```

- ① Create HTTP server, using anonymous function to define per-request behavior
- ② Determine HTML file to be served by default
- ③ Translate URL path to relative file path
- ④ Serve the static file

STARTING THE HTTP SERVER

While we've created the HTTP server in the code, we haven't actually added the logic needed to start it. Add the following lines which start the server, requesting that it listen on TCP/IP port 3000. Port 3000 is an arbitrary choice: any unused port above 1024 would work as well (a port under 1024 may also work if you're running Windows or, if in Linux or OS X, start your application using a privileged user such as "root").

```
server.listen(3000, function() {
```

```
    console.log("Server listening on port 3000.");
});
```

If you'd like to see what the application can do at this point you can start the server by entering the following into your command-line prompt:

```
node server.js
```

With the server running, visiting `http://127.0.0.1:3000` in your web browser will simply result in the triggering of the 404 error helper and "Error 404: resource not found." will be displayed. While you've added static file handling logic, you haven't added the static files themselves.

A running server can be stopped by using `CTRL+C` on the command-line.

Next, let's move on to adding the static files necessary to get the chat application more functional.

2.3.2 Adding the HTML and CSS files

The first static file you'll add is the base HTML. Create a file in the "public" directory named "index.html" and place the HTML in listing 2.5 in it. The HTML will include a CSS file, set up some HTML DIV elements in which application content will be displayed, and will load a number of client-side JavaScript files. The JavaScript files provide client-side Socket.io functionality, jQuery (for easy DOM manipulation), and a couple of application-specific files providing chat functionality.

Listing 2.5 public/index.html: The HTML for the chat application

```
<!doctype html>
<html lang='en'>

<head>
  <title>Chat</title>
  <link rel='stylesheet' href='/stylesheets/style.css'></link>
</head>

<body>
<div id='content'>
  <div id='room'></div> ①
  <div id='room-list'></div> ②
  <div id='messages'></div> ③
</div>

```

```

<form id='send-form'>
  <input id='send-message' /> ④
  <input id='send-button' type='submit' value='Send' />

  <div id='help'>
    Chat commands:
    <ul>
      <li>Change nickname: <code>/nick [username]</code></li>
      <li>Join/create room: <code>/join [room name]</code></li>
    </ul>
  </div>
</form>
</div>

<script src='/socket.io/socket.io.js' type='text/javascript'></script>
<script src='http://code.jquery.com/jquery-1.8.0.min.js' type='text/javascript'></script>
<script src='/javascripts/chat.js' type='text/javascript'></script>
<script src='/javascripts/chat_ui.js' type='text/javascript'></script>
</body>
</html>

```

- ① DIV in which the current room name will be displayed
- ② DIV in which a list of available rooms will be displayed
- ③ DIV in which chat messages will be displayed
- ④ Form input element in which the user will enter commands and messages

The next file we'll add defines the application's CSS styling. In the "public/stylesheets" directory create a file named "style.css" and put the CSS code in listing 2.6 in it.

Listing 2.6 public/stylesheets/style.css: Application CSS

```

body {
  padding: 50px;
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;
}

a {
  color: #00B7FF;
}

#content { ①
  width: 800px;
  margin-left: auto;
  margin-right: auto;
}

#room { ②
  background-color: #ddd;
}

```

```

margin-bottom: 1em;
}

#messages { ③
  width: 690px;
  height: 300px;
  overflow: auto; ④
  background-color: #eee;
  margin-bottom: 1em;
  margin-right: 10px;
}

```

- ① The application will be 800 pixels wide and horizontally centered
- ② CSS rules for area in which current room name is displayed
- ③ The message display area will be 690 pixels wide and 300 pixels high
- ④ Allow DIV in which messages are displayed to scroll when it's filled up with content

With the HTML and CSS roughed out, run the application and take a look using your web browser. The application should look like figure 2.9.

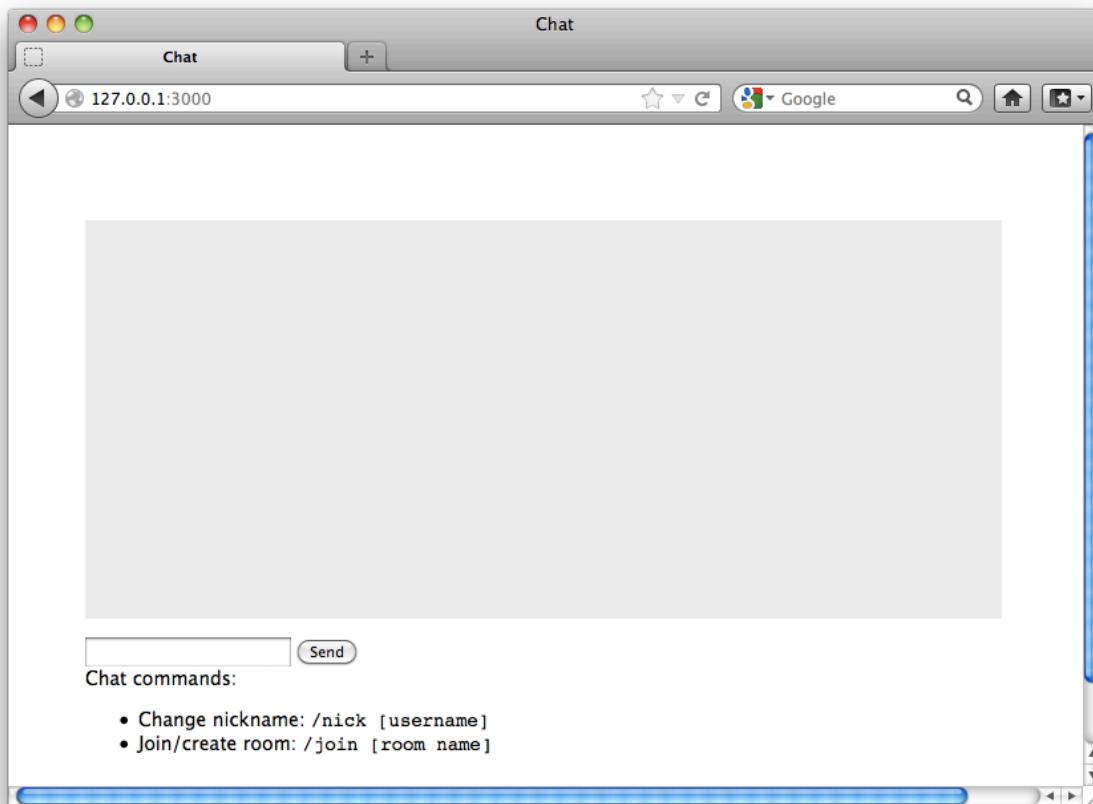


Figure 2.9 The application-in-progress.

The application isn't yet functional, but static files are being served and the

basic visual layout is established. With that taken care of, let's move on to defining the server-side chat message dispatching.

2.4 Handling chat-related messaging using *Socket.io*

Of the three things we said that the app had to do, we've already covered the first one, serving static files - and now we're going to tackle the second – handling communication between the browser and server. Modern browsers⁶ are capable of using WebSocket to handle communication between the browser and the server.

Footnote 6 <http://socket.io/#browser-support>

Socket.io provides a layer of abstraction over WebSocket and other transports for both Node and client-side JavaScript. It will fallback transparently to other WebSocket alternatives if it's not implemented in a web browser while keeping the same API. In this section, we'll:

- briefly introduce you to socket.io and define the socket.io functionality you'll need on the server-side
- add code that sets up a socket.io server
- add code to handle various chat application events

Socket.io, out of the box, provides virtual "channels" so instead of broadcasting every message to every connected user you can broadcast only to those who are "subscribed" to a specific channel. This functionality makes implementing chat rooms in our application quite simple, as you'll see later.

Socket.io is also a good example of the usefulness of what are called "event emitters" in Node. Event emitters are, in essence, a handy design pattern for organizing asynchronous logic. While you'll see some event emitter code at work in this chapter, we'll go into more detail in the next chapter.

SIDE BAR

Event emitters

An event emitter is associated with a conceptual resource of some kind and can send and receive messages to and from the resource. Some examples of what the "resource" could be include a connection to a remote server or something more abstract, like a game character.

The Johnny-Five project⁷, in fact, leverages Node for robotics applications, using event emitters to control Arduino microcontrollers.

Footnote 7 <https://github.com/rwldrn/johnny-five>

First, we'll start the server functionality and establish connection logic. Then,

we'll define the functionality you need on the server side.

2.4.1 Setting up the Socket.io server

To begin, append the following two lines to `server.js`. The first line loads functionality from a custom Node module that we'll define next. This module supplies logic to handle Socket.io-based server-side chat functionality. The next line starts the Socket.io server functionality, providing it with an already defined HTTP server so it can share the same TCP/IP port.

```
var chatServer = require('./lib/chat_server');
chatServer.listen(server);
```

You now need to create a new file, `chat_server.js`, inside the `lib` directory. Start this file by adding the following variable declarations. These declarations allow the use of Socket.io and initialize a number of variables that define chat state.

```
var socketio = require('socket.io');
var io;
var guestNumber = 1;
var nickNames = {};
var namesUsed = [];
var currentRoom = {};
```

ESTABLISHING CONNECTION LOGIC

Next, add the logic in listing 2.7 to define the chat server function `listen`. This function is invoked in `server.js`. It starts the Socket.io server, limits the verbosity of Socket.io's logging to console, and establishes how each incoming connection should be handled.

The connection handling logic, you'll notice, calls a number of helper functions that we'll add next to `chat_server.js`.

Listing 2.7 lib/chat_server.js: Starting up a Socket.io server

```
exports.listen = function(server) {
  io = socketio.listen(server); ①
  io.set('log level', 1);
```

```

io.sockets.on('connection', function (socket) { ②
  guestNumber = assignGuestName(socket, guestNumber,
    ↗ nickNames, namesUsed); ③
  joinRoom(socket, 'Lobby'); ④

  handleMessageBroadcasting(socket, nickNames); ⑤
  handleNameChangeAttempts(socket, nickNames, namesUsed);
  handleRoomJoining(socket);

  socket.on('rooms', function() { ⑥
    socket.emit('rooms', io.sockets.manager.rooms);
  });
}

handleClientDisconnection(socket, nickNames, namesUsed); ⑦
});
};

```

- ① Start the Socket.io server, allowing it to piggyback on the existing HTTP server
- ② Define how each user connection will be handled
- ③ Assign user a guest name when they connect
- ④ Place user in the "Lobby" room when they connect
- ⑤ Handle user messages, name change attempts, and room creation/changes.
- ⑥ Provide user with a list of occupied rooms on request.
- ⑦ Define "cleanup" logic for when a user disconnects

With the connection handling established, you'll now add the individual helper functions that will handle the application's needs.

2.4.2 Handling application scenarios and events

The chat application needs to handle the following types of scenarios and events:

- guest name assignment
- name change requests
- chat messages
- room creation
- user disconnection

ASSIGNING GUEST NAMES

The first helper function you need to add is `assignGuestName`, which handles the naming of new users. When a user first connects to the chat server, the user is placed in a chat room named "Lobby" and `assignGuestName` is called to assign them a name to distinguish them from other users.

Each guest name is essentially the word "Guest" followed by a number that

increments each time a new user connects. The guest name is stored in the nickNames variable for reference, associated with the internal socket ID. The guest name is also added to namesUsed, a variable in which names that are being used are stored. Add the code in listing 2.8 to lib/chat_server.js to implement this functionality.

Listing 2.8 lib/chat_server.js: Assigning a guest name

```
function assignGuestName(socket, guestNumber, nickNames, namesUsed) {
    var name = 'Guest' + guestNumber; 1
    nickNames[socket.id] = name; 2
    socket.emit('nameResult', { 3
        success: true,
        name: name
    });
    namesUsed.push(name); 4
    return guestNumber + 1; 5
}
```

- 1** Generate new guest name
- 2** Associate guest name with client connection ID
- 3** Let user know their guest name
- 4** Note that guest name is now used
- 5** Increment counter used to generate guest names

JOINING ROOMS

The second helper function you'll need to add is `joinRoom`. This function, shown in listing 2.9, handles logic related to a user joining a chat room.

While having a user join a Socket.io room is simple, requiring only a call to the `join` method of a socket object, the application communicates related details to the user and other users in the same room. The application lets the user know what other users are in the room and lets these other users know that the user is now present.

Listing 2.9 lib/chat_server.js: Logic related to joining a room

```
function joinRoom(socket, room) {
    socket.join(room); 1
    currentRoom[socket.id] = room; 2
```

```

socket.emit('joinResult', {room: room}); ③
socket.broadcast.to(room).emit('message', {
  text: nickNames[socket.id] + ' has joined ' + room + '.';
});

var usersInRoom = io.sockets.clients(room); ⑤
if (usersInRoom.length > 1) { ⑥
  var usersInRoomSummary = 'Users currently in ' + room + ': ';
  for (var index in usersInRoom) {
    var userSocketId = usersInRoom[index].id;
    if (userSocketId != socket.id) {
      if (index > 0) {
        usersInRoomSummary += ', ';
      }
      usersInRoomSummary += nickNames[userSocketId];
    }
  }
  usersInRoomSummary += '.';
  socket.emit('message', {text: usersInRoomSummary}); ⑦
}
}

```

- ① Make user join room
- ② Note that user is now in this room
- ③ Let user know they're now in a new room
- ④ Let other users in room know that a user has joined
- ⑤ Determine what other users are in the same room as the user
- ⑥ If other users exist, summarize who they are
- ⑦ Send the summary of other users in the room to the user

HANDLING NAME CHANGE REQUESTS

If every user just kept their guest name, it would be hard to remember who's who. For this reason the chat application allows the user to request a name change. As figure 2.10 shows, a name change involves the user's web browser making a request via Socket.io then receiving a response indicating success or failure.

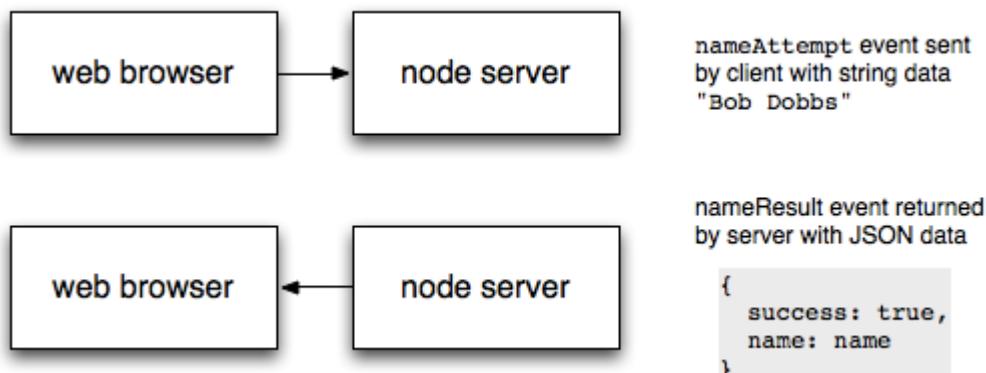


Figure 2.10 A name change request and response

Add the code in listing 2.10 to `lib/chat_server.js` to define a function that handles requests by users to change their names. From the application's perspective, the users aren't allowed to change their name to anything beginning with "Guest" or use a name that's already in use.

Listing 2.10 lib/chat_server.js: Logic to handle name request attempts

```
function handleNameChangeAttempts(socket, nickNames, namesUsed) {
    socket.on('nameAttempt', function(name) { ①
        if (name.indexOf('Guest') == 0) { ②
            socket.emit('nameResult', {
                success: false,
                message: 'Names cannot begin with "Guest".'
            });
        } else {
            if (namesUsed.indexOf(name) == -1) { ③
                var previousName = nickNames[socket.id];
                var previousNameIndex = namesUsed.indexOf(previousName);
                namesUsed.push(name);
                nickNames[socket.id] = name;
                delete namesUsed[previousNameIndex]; ④
                socket.emit('nameResult', {
                    success: true,
                    name: name
                });
                socket.broadcast.to(currentRoom[socket.id]).emit('message', {
                    text: previousName + ' is now known as ' + name + '.'
                });
            } else {
                socket.emit('nameResult', { ⑤
                    success: false,
                    message: 'That name is already in use.'
                });
            }
        }
    });
}
```

- ① Added listener for nameAttempt events
- ② Don't allow nicknames to begin with "Guest"
- ③ If the name isn't already registered, register it
- ④ Remove previous name to make available to other clients.
- ⑤ Send an error to the client if the name's already registered

SENDING CHAT MESSAGES

Now that user nicknames are taken care of, you'll add a function that defines how a chat message sent from a user is handled. Figure 2.11 shows the basic process: the user emits an event indicating the room where the message is to be sent and the message text. The server then relays the message to all other users in the same room.

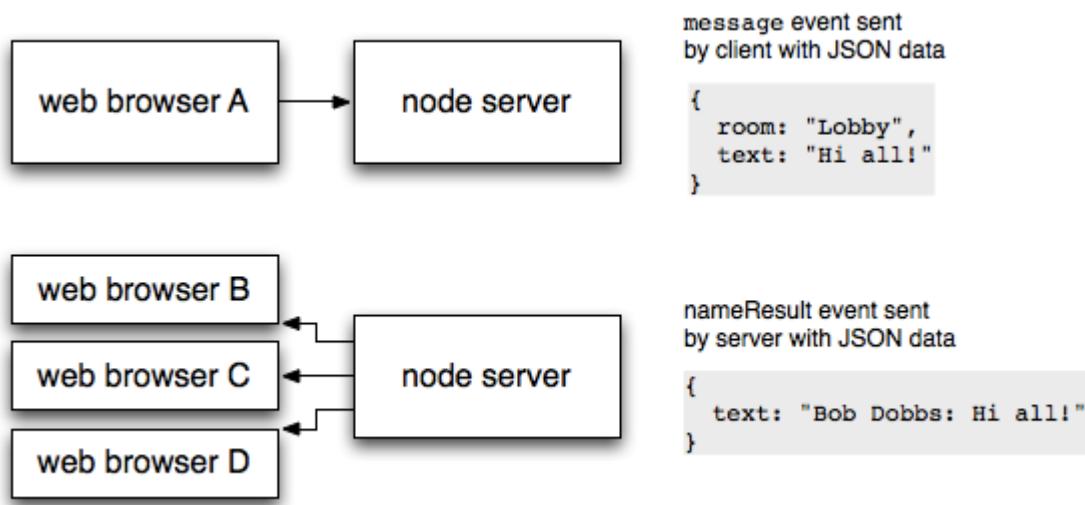


Figure 2.11 Sending a chat message

Add the following code to `lib/chat_server.js`. Socket.io's `broadcast` function is used to relay the message.

```

function handleMessageBroadcasting(socket) {
  socket.on('message', function (message) {
    socket.broadcast.to(message.room).emit('message', {
      text: nickNames[socket.id] + ': ' + message.text
    });
  });
}

```

CREATING ROOMS

Next, you'll add functionality that allows a user to join an existing room or, if it doesn't yet exist, create it. Figure 2.12 shows visually the interaction between the user and the server.

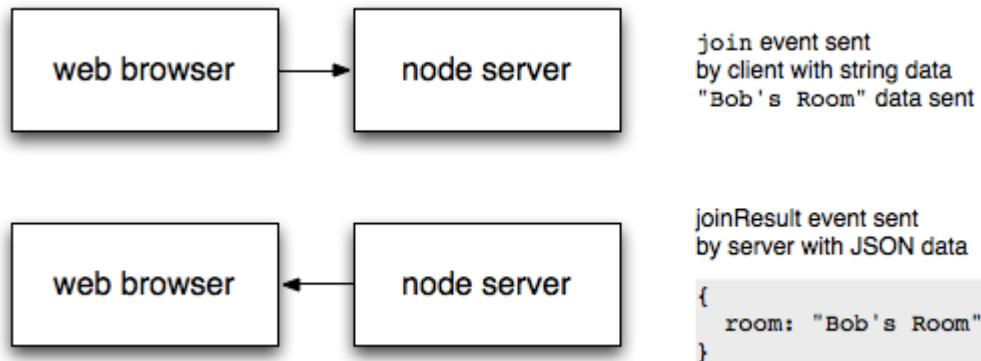


Figure 2.12 Changing to a different chat room

Add the following code to `lib/chat_server.js` to enable room changing. Note the use of Socket.io's `leave` method.

```

function handleRoomJoining(socket) {
  socket.on('join', function(room) {
    socket.leave(currentRoom[socket.id]);
    joinRoom(socket, room.newRoom);
  });
}

```

HANDLING USER DISCONNECTIONS

Finally add the logic below to `lib/chat_server.js` to remove a user's nickname from `nickNames` and `namesUsed` when the user leaves the chat application.

```

function handleClientDisconnection(socket) {
  socket.on('disconnect', function() {
    var nameIndex = namesUsed.indexOf(nickNames[socket.id]);
    delete namesUsed[nameIndex];
    delete nickNames[socket.id];
  });
}

```

With the server-side components fully defined, you're now ready to further develop the client-side logic.

2.5 Using client-side JavaScript for the application's user interface

Now that you've added server-side Socket.io logic to dispatch messages sent from the browser, let's add the client-side JavaScript needed to communicate with the server.

Client-side Javascript is needed to handle the following functionality:

- determine whether user input is a message or a chat command
- process chat commands and send information to the server

Let's start with the first piece of functionality.

2.5.1 Relaying messages and name/room changes to the server

The first bit of client-side JavaScript you'll add is a JavaScript prototype object that will process chat commands, send messages, and request room and nickname changes.

In the "public/javascripts" directory create a file named "chat.js" and put the following code in it. This code starts JavaScript's equivalent of a "class" that takes a single argument, a Socket.io socket, when instantiated.

```
var Chat = function(socket) {
  this.socket = socket;
};
```

Next, add the following function to send chat messages.

```
Chat.prototype.sendMessage = function(room, text) {
  var message = {
    room: room,
    text: text
  };
  this.socket.emit('message', message);
};
```

Add the following function to change rooms.

```
Chat.prototype.changeRoom = function(room) {
  this.socket.emit('join', {
```

```

    newRoom: room
  });
};

}

```

Finally, add the function defined in listing 2.11 for processing a chat command. Two chat commands are recognized: "join" for joining/creation a room and "nick" for changing one's nickname.

Listing 2.11 public/javascripts/chat.js: Processing chat commands

```

Chat.prototype.processCommand = function(command) {
  var words = command.split(' ');
  var command = words[0]
    .substring(1, words[0].length)
    .toLowerCase(); 1
  var message = false;

  switch(command) {
    case 'join':
      words.shift();
      var room = words.join(' ');
      this.changeRoom(room); 2
      break;

    case 'nick':
      words.shift();
      var name = words.join(' ');
      this.socket.emit('nameAttempt', name); 3
      break;

    default:
      message = 'Unrecognized command.'; 4
      break;
  }

  return message;
};

```

- 1** Parse command from first word
- 2** Handle room changing/creation
- 3** Handle name change attempts
- 4** Return an error message if the command isn't recognized

2.5.2 Showing messages and available rooms in the user interface

You'll now start adding logic that interacts directly with the browser-based user interface using jQuery. The first functionality you'll add will be to display text data.

In web applications there are, from a security perspective, two types of text data. There is "trusted" text data, which consists of text supplied by the application, and "untrusted" text data, which is text created by, or derived from, users of the application. Text data from users is considered untrusted because malicious users may intentionally submit text data that includes JavaScript logic contained in `<script>` tags. This text data, if displayed unaltered to other users, can cause nasty things to happen, like redirecting users to another web page. This method of hijacking a web application is called a cross-site scripting (XSS) attack.

The chat application will use two helper functions to display text data. One function will display untrusted text data and the other function will display trusted text data.

The function `divEscapedContentElement` will handle displaying untrusted text. `divEscapedContentElement` will sanitize text by transforming special characters into HTML entities, as shown in figure 2.13, so the browser knows to display them as entered rather than attempt to interpret them as part of an HTML tag.

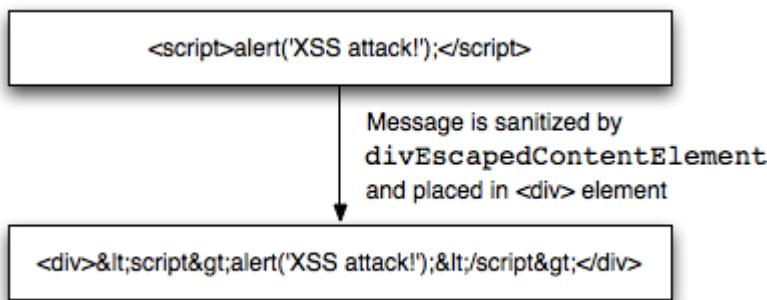


Figure 2.13 Escaping untrusted content

The function `divSystemContentElement` will display trusted content created by the system rather than other users.

In the "public/javascripts" directory add a file named "chat_ui.js" and put the following two helper functions in it.

```

function divEscapedContentElement(message) {
    return $('

</div>').text(message);
}

function divSystemContentElement(message) {
    return $('

</div>').html('<i>' + message + '</i>');
}


```

The next function you'll append to "chat_ui.js", detailed in listing 2.12, is for processing user input. If user input begins with the "/" character, it is treated as a chat command. If not, it is sent to the server, as a chat message to be broadcast to other users, and added to the chat room text of the room the user's currently in.

Listing 2.12 chat_ui.js: processing raw user input

```

function processUserInput(chatApp, socket) {
    var message = $('#send-message').val();
    var systemMessage;

    if (message.charAt(0) == '/') { ①
        systemMessage = chatApp.processCommand(message);
        if (systemMessage) {
            $('#messages').append(divSystemContentElement(systemMessage));
        }
    } else {
        chatApp.sendMessage($('#room').text(), message); ②
        $('#messages').append(divEscapedContentElement(message));
        $('#messages').scrollTop($('#messages').prop('scrollHeight'));
    }

    $('#send-message').val('');
}

```

- ① If user input begins with a slash, treat it as a command
- ② Broadcast non-command input to other users

Now that you've got some helper functions defined, you'll add the logic in listing 2.13 which is meant to execute when the web page has fully loaded in the user's browser. This code handles client-side initiation of Socket.io event handling.

Listing 2.13 chat_ui.js: client-side application initialization logic

```

var socket = io.connect();

$(document).ready(function() {

```

```

var chatApp = new Chat(socket);

socket.on('nameResult', function(result) { ①
    var message;

    if (result.success) {
        message = 'You are now known as ' + result.name + '.';
    } else {
        message = result.message;
    }
    $('#messages').append(divSystemContentElement(message));
});

socket.on('joinResult', function(result) { ②
    $('#room').text(result.room);
    $('#messages').append(divSystemContentElement('Room changed.'));
});

socket.on('message', function (message) { ③
    var newElement = $('').text(message.text);
    $('#messages').append(newElement);
});

socket.on('rooms', function(rooms) { ④
    $('#room-list').empty();

    for(var room in rooms) {
        room = room.substring(1, room.length);
        if (room != '') {
            $('#room-list').append(divEscapedContentElement(room));
        }
    }
});

$('#room-list div').click(function() { ⑤
    chatApp.processCommand('/join ' + $(this).text());
    $('#send-message').focus();
});

setInterval(function() { ⑥
    socket.emit('rooms');
}, 1000);

$('#send-message').focus();

$('#send-form').submit(function() {
    processUserInput(chatApp, socket);
    return false;
});
});

```

- ① Display the results of a name change attempt
- ②

- Display the results of a room change
- ③ Display received messages
- ④ Display list of rooms available
- ⑤ Allow the click of a room name to change to that room
- ⑥ Request list of rooms available intermittantly
- ⑦ Allow clicking the send button to send a chat message

To finish the application off, add the final CSS styling code in listing 2.14 to the "public/stylesheets/style.css" file.

Listing 2.14 public/stylesheets/style.css: Final additions to style.css

```
#room-list {
    float: right;
    width: 100px;
    height: 300px;
    overflow: auto;
}

#room-list div {
    border-bottom: 1px solid #eee;
}

#room-list div:hover {
    background-color: #ddd;
}

#send-message {
    width: 700px;
    margin-bottom: 1em;
    margin-right: 1em;
}

#help {
    font: 10px "Lucida Grande", Helvetica, Arial, sans-serif;
}
```

With the final code added, try running the application (using `node server.js`). It should look like figure 2.14.

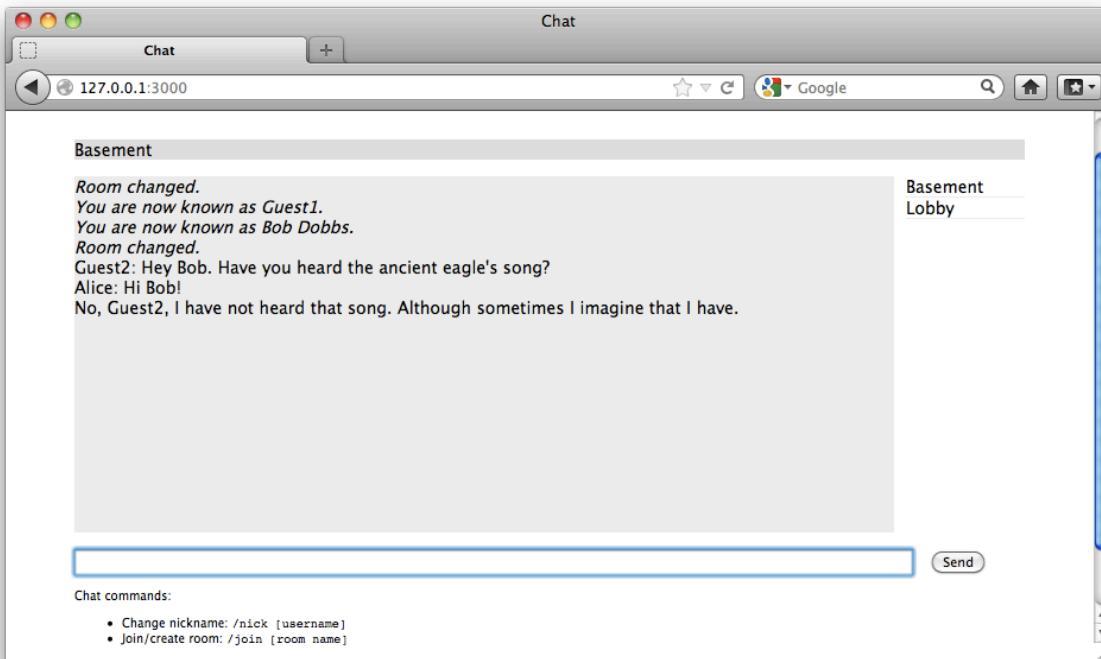


Figure 2.14 The completed chat application.

2.6 Summary

You've now completed a small real-time web application using Node.js!

You should have a sense of how the application is constructed and what the code is like. If aspects of this example application are unclear, don't worry: in the following chapters we'll go into depth on the techniques and technologies used in this example.

Before you delve into the specifics of Node development, however, you'll want to learn how to deal with the unique challenges of asynchronous development. The next chapter will teach you essential techniques and tricks that will save you a lot of time and frustration.



Node programming fundamentals

In this chapter:

- Organizing your code into modules
- Coding conventions
- Handling one-off events with callbacks
- Handling repeating events with event emitters
- Implementing serial and parallel control flow
- Leveraging flow control tools

Node, unlike many open source web frameworks, is easy to set up and doesn't require much in terms of memory and disk space. No complex integrated development environments or byzantine build systems are required. Some fundamental knowledge will, however, help you a lot when starting out.

In this chapter we'll address two challenges that new Node developers face:

- how to organize your code
- how asynchronous programming works

The problem of organizing code is familiar to most experienced programmers. Logic is organized conceptually into classes and functions. Files containing the classes and functions are organized into directories within the source tree. In the end, code is organized into applications and libraries. Node's module system provides a powerful mechanism for organizing your code and you'll learn how to harness it in this chapter.

Asynchronous programming will likely take some time to grasp and master as it

requires a paradigm shift in terms of thinking of how application logic should execute. With synchronous programming, you can write a line of code knowing that all the lines of code that came before it will have executed already. With asynchronous development, however, application logic can initially seem like a Rube Goldberg machine¹. It's worth taking the time, before beginning development of a large project, to learn how you can elegantly control your application's behavior.

Footnote 1 http://en.wikipedia.org/wiki/Rube_Goldberg_machine

In this chapter you'll learn a number of important asynchronous programming techniques that will allow you to keep a tight rein on how your application executes. You're going to learn how to respond to one-time events, how to handle repeating events, and how to sequence asynchronous logic.

Let's start, however, with how you can tackle the problem of code organization through the use of "modules", Node's way of keeping code organized and packaged for easy reuse.

3.1 Organizing and reusing Node functionality

When creating an application, Node or otherwise, there often comes a point where putting all of your code in a single file becomes unwieldy. Once this happens, the conventional approach, as represented visually in figure 3.1, is to take a file containing a lot of code and try to organize it by grouping related logic and moving it into separate files.

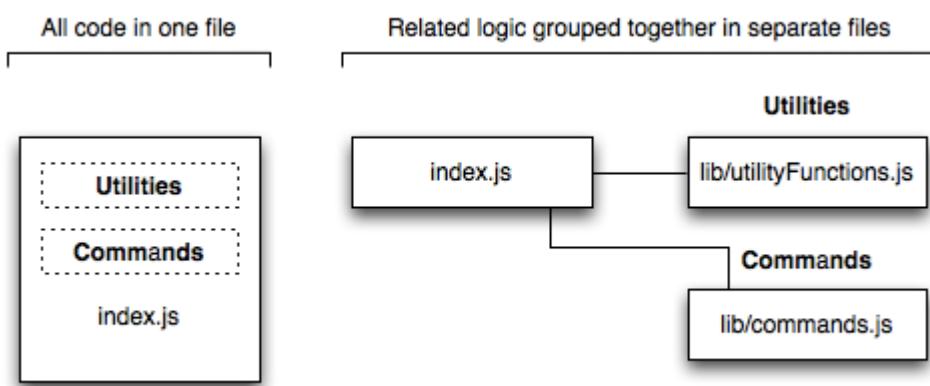


Figure 3.1 It's easier to navigate your code if you organize it using directories and separate files rather than keeping your application in one long file.

In some language implementations, such as PHP and Ruby, incorporating the logic from another file (we'll call this the "included" file) can mean all the logic

executed in the included file affects the global scope. This means that any variables created and functions declared in the included file risk overwriting those created and declared by the application.

Say you were programming in PHP. Your application might contain the following logic:

```
function uppercase_trim($text) {
    return trim(strtoupper($text));
}

include('string_handlers.php');
```

If your `string_handlers.php` file also attempted to define a `uppercase_trim` function you'd receive the following error:

```
Fatal error: Cannot redeclare uppercase_trim()
```

In PHP you can avoid this by using "namespaces"² and Ruby offers similar functionality through "modules"³. Node, however, avoids the possibility by not offering an easy way to accidentally pollute the global namespace.

Footnote 2 <http://php.net/manual/en/language.namespaces.php>

Footnote 3 <http://www.ruby-doc.org/core-1.9.3/Module.html>

Node modules bundle up code for reuse, but don't alter global scope. They, in fact, allow you to select what functions and variables from the included file are exposed to the application. If returning more than one function/variable then a module can specify these by setting the properties of an object called "exports". If returning a single function/variable then "module.exports" can, instead, be set. Figure 3.2 shows how this works visually.

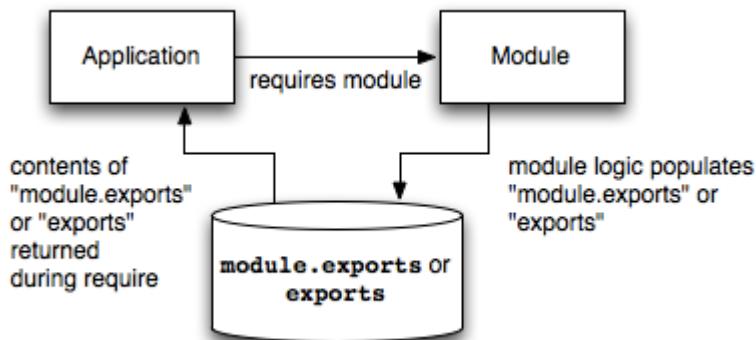


Figure 3.2 The population of "module.exports" or the "exports" object allows a module to select what should be shared with the application.

If this seems a bit confusing, don't worry: we'll run through a number of examples in this chapter.

By avoiding pollution of the global scope, Node's module system avoids naming conflicts and simplifies code reuse. Modules can then be published to the npm (Node Package Manager) repository, an online collection of ready-to-use Node modules, and shared with the Node community without those using the modules having to worry about one module overwriting the variables and functions of another. We'll talk about how to publish to the npm repository in chapter 13.

To help you organize your logic into modules, we'll explain how you can create modules, where modules are stored in the filesystem, and things to be aware of when creating and using modules.

3.1.1 Creating modules

Modules can either be single files or directories containing one or more files as can be seen on figure 3.3. If a module is a directory, the file in the module directory that will be evaluated is normally named "index.js" (although this can be overridden: see Section 3.1.4 "Caveats").



Figure 3.3 Node modules can either be created using files (example 1) or directories (example 2).

To create a typical module, you create a file that defines properties on the `exports` object with any kind of data, such as strings, objects, and functions.

If you wanted to create an application that dealt with converting between a

number of different currencies, you might look for a community-created module that had this functionality. If you couldn't find what you needed, you might write something to fill the need, possibly contributing it back to the community.

To show how a basic module is created, let's add some currency conversion functionality to a file named "currency.js". This file will contain two functions that will convert Canadian dollars to US dollars, and vice versa:

Listing 3.1 Defining a Node module

```
var canadianDollar = 0.91;

function roundTwoDecimals(amount) {
    return Math.round(amount * 100) / 100;
}

exports.canadianToUS = function(canadian) {❶
    return roundTwoDecimals(canadian * canadianDollar);
}

exports.USToCanadian = function(us) {❷
    return roundTwoDecimals(us / canadianDollar);
}
```

- ❶ The `canadianToUS` function is set in the `exports` module so it can be used by code requiring this module
- ❷ The `USToCanadian` function is also set in the `exports` module

Note that only two properties of the `exports` object are set. This means only the two functions, `canadianToUS` and `USToCanadian` can be accessed by the application including the module. The variable `canadianDollar` acts as a private variable that effects the logic in `canadianToUS` and `USToCanadian` but can't be directly accessed by the application.

To utilize your new module, you can use Node's `require` function, which takes as an argument a path to the module you wish to use. Node performs a synchronous lookup in order to locate, and load the file's contents. Because `require` is synchronous, unlike most functions in the node API, you do not need to supply `require` with a callback function.

In the following code you `require` the "currency.js" module.

Listing 3.2 test-currency.js: Requiring a module

```

var currency = require('./currency');

console.log('50 Canadian dollars equals this amount of US dollars:');
console.log(currency.canadianToUS(50)); 1

console.log('30 US dollars equals this amount of Canadian dollars:');
console.log(currency.USToCanadian(30)); 2 3

```

- ➊ The path begins with "./" to indicate that the module exists within the same directory as the application script
- ➋ Make use of the currency module's canadianUtoUS function
- ➌ Make use of the currency module's USToCanadian function

Requiring a module that begins with "./" means that if you were to create your application script named "test-currency.js" in a directory named "currency_app", then your "currency.js" module file, as represented visually in figure 2.5, would also need to exist in the "currency_app" directory. When requiring, the ".js" extension is assumed, so you can omit it if desired.

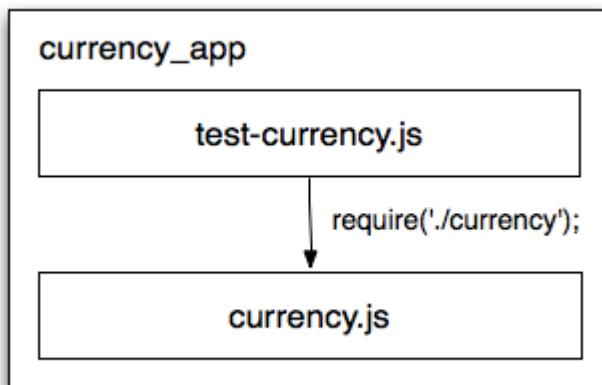


Figure 3.4 When putting a "./" at the beginning of a module require, Node will look in the same directory as the program file being executed.

After node has located and evaluated your module, the `require` function returns the contents of "exports" defined in the module. You're then able to use the two functions returned by the module to do currency conversion.

If you wanted to put the module into a subdirectory, "lib" for example, you could do so by simply changing the line containing the `require` logic to the following.

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

Populating the `exports` object of a module gives you a simple way to group reusable code in separate files.

3.1.2 Fine Tuning Module Creation using `module.exports`

While populating the `exports` object with functions and variables is suitable for most module creation needs, there will be times when you want a module to deviate from this model.

The currency convertor module created earlier in this section, for example, could be redone to return a JavaScript class, rather than an object containing functions, as shown in listing 3.1. An object-oriented implementation could behave something like the following.

```
var Currency = require('./currency')
, canadianDollar = 0.91;

currency = new Currency(canadianDollar);
console.log(currency.canadianToUS(50));
```

By returning a class from `require`, rather than an object, our code can be more elegant if a class is the only thing we need from the module. To create a module that will return anything other than an object you might guess that you simply need to set `exports` to whatever you want to return. This, however, won't work as Node expects `exports` to remain an object. The following module code attempts to set `exports` to a class and, when you try to use it with the previous code, a `TypeError: object is not a function` exception will be raised.

Listing 3.3 incorrect_module.js This module won't work as expected

```
var Currency = function(canadianDollar) {
  this.canadianDollar = canadianDollar;
}

Currency.prototype.roundTwoDecimals = function(amount) {
  return Math.round(amount * 100) / 100;
```

```

}

Currency.prototype.canadianToUS = function(canadian) {
  return this.roundTwoDecimals(canadian * this.canadianDollar);
}

Currency.prototype.USToCanadian = function(us) {
  return this.roundTwoDecimals(us / this.canadianDollar);
}

exports = Currency;

```

In order to get the previous module code to work as expected, you'd need to replace `exports` with `module.exports`. The `module.exports` mechanism is provided as a way to work around Node's inability to set `exports` to something other than an object. If you create a module that populates both `exports` and `module.exports`, `module.exports` will be returned and `exports` will be ignored.

By using either `exports` and `module.exports`, depending on our needs, we can organize functionality into modules and avoid the pitfall of ever-growing application scripts.

3.1.3 Reusing modules using the "node_modules" folder

Requiring modules that exist relative, in the filesystem, to an application is useful for organizing application-specific code, but isn't as useful for code you'd like to reuse between applications or share with others. For code reuse Node includes a unique mechanism that allows modules to be required without knowing their location in the filesystem. This mechanism is the use of "node_modules" directories.

In the earlier module example, we required `"/./currency"`. If you omit the `"/."` and simply require `"currency"` Node will follow a number of rules, as specified in figure 1.5, with which it will search for this module.

Figure 3.5 Steps to finding a module

The `NODE_PATH` environmental variable provides a way to specify alternative locations for Node modules. If used, `NODE_PATH` should be set to a list of directories, separated by semi-colons in Windows or colons in other operating systems.

3.1.4 Caveats

While the essence of Node's module system is straightforward, there are two things to be aware of.

First, if a module is a directory, the file in the module directory that will be evaluated must be named "index.js", unless specified otherwise by a file in the module directory named "package.json". To specify an alternative to "index.js", the "package.json" file must contain JavaScript Object Notation (JSON) data defining an object with a key named "main" that specifies the path, within the module directory, to the main file. Figure 2.6 shows a flow chart summarizing these rules.

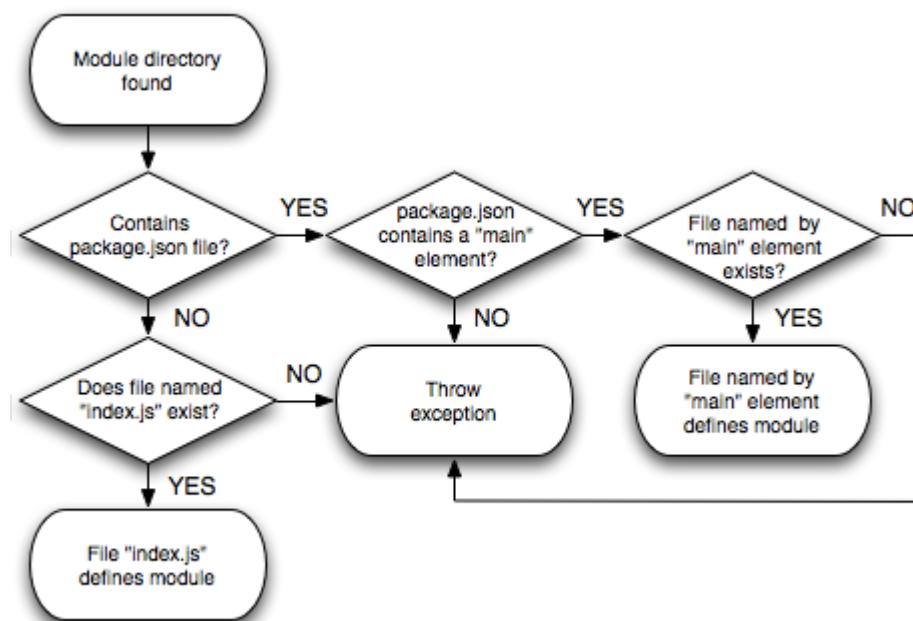


Figure 3.6 The "package.json" file, when placed in a module directory, allows you to define your module using a file other than "index.js".

Below is an example of a "package.json" file specifying that "currency.js" is the main file.

```
{
  "main": "./currency.js"
}
```

The other thing to be aware of is Node's ability to cache modules as objects. If two files in an application require the same module, the first require will store the

data returned in application memory so the second require won't need to access and evaluate the module's source files. The second require will, in fact, have the opportunity to alter the cached data. This "monkey patching" capability allows one module to modify the behaviour of another, freeing the developer from having to create a new version of it.

The best way to get comfortable with Node's module system is to play with it, verifying the behavior described in this section yourself. Now that you have a basic understanding of how modules work, let's move on to asynchronous programming techniques.

3.2 Asynchronous development challenges

When creating asynchronous applications you have to pay close attention to how your application flows and keep a watchful eye on application state: the conditions of the event loop, application variables, and any other resources that change as program logic executes.

Node's event loop, for example, keeps track of asynchronous logic that hasn't completed processing. As long as there is uncompleted asynchronous logic, the Node process won't exit. A continually running Node process is desirable behavior for something like a web server, but it isn't desirable for applications like command-line tools. The event loop will keep track of any database connections until they're closed, preventing Node from exiting.

Application variables can also change unexpectedly if you're not careful. Listing 3.1 shows an example of how the order in which asynchronous code executes can lead to confusion. If the example code was executing synchronously you'd expect the output to be "The color is blue." As the example is asynchronous, however, the value of the `color` variable changes before `console.log` executes and the output is "The color is green."

Listing 3.4 scope_behavior.js: an example of how scope behavior can leads to bugs

```
function asyncFunction(callback) {
  setTimeout(function() {
    callback()
  }, 200);
}

var color = 'blue';
```

```

asyncFunction(function() {
  console.log('The color is ' + color);
});

color = 'green';

```

To "freeze" the contents of the `color` variable you can modify your logic and use a JavaScript closure. In Listing 3.2, you wrap the call to `asyncFunction` in an anonymous function that takes a `color` argument. You then immediately execute the anonymous function, sending it the current contents of `color`. By making `color` an argument for the anonymous function, it becomes local to the scope of that function and when the value of `color` is changed outside of the anonymous function, the local version is unaffected.

Listing 3.5 closure_trick.js: an example of using an anonymous function to preserve a global variable's value

```

function asyncFunction(callback) {
  setTimeout(function() {
    callback()
  }, 200);
}

var color = 'blue';

(function(color) {
  asyncFunction(function() {
    console.log('The color is ' + color);
  })
})(color);

color = 'green';

```

This is but one of many JavaScript programming tricks you'll come across during Node development.

Now that you understand how to keep the event loop free and control your application state, let's look at how to cleanly structure your asynchronous application logic.

3.3 Asynchronous programming techniques

If you've done front-end web programming in which interface events (such as mouse clicks) trigger logic then you've done asynchronous programming. Server-side asynchronous programming is no different: events occur that trigger response logic. There are two popular models in the Node world for managing response logic: callbacks and event listeners.

Callbacks generally define logic for "one-off" responses. If you perform a database query, for example, you can specify a callback to determine what to do with the query results. The callback may display the database results, do a calculation based on the results, or execute another callback using the query results as an argument.

Event listeners, on the other hand, are essentially callbacks that are associated with a conceptual entity. The Node `events` module allows the creation of "event emitters" to which listeners can be added. The name "emitter" implies that event emitters are mainly for sending events, but they receive as well as send. Listeners for different types of events can be added, with each event type having a name. Events can then be triggered, using the event type name, by the event emitter. Node's HTTP server is implemented using event emitters to handle requests with an event listener handling response logic.

Now that we've established that response logic is generally organized in one of two ways in Node, let's jump into how it all works by learning first how to handle one-off events with callbacks, then how to respond to repeating events using event emitter listeners.

3.3.1 Handling one-off events with callbacks

A callback is a function, passed as an argument to an asynchronous function, that describes what to do after the asynchronous operation has completed. Callbacks are used frequently in Node development, more so than event emitters, and as you will see in the following example, callbacks are comparatively simple to use.

As an example of the use of callbacks in an application, let's make a simple HTTP server that pulls the title of the ten most recent posts stored in a text file containing JSON, assembles an HTML page containing the titles, then sends the HTML page to the user. The results will be similar to figure 3.1.

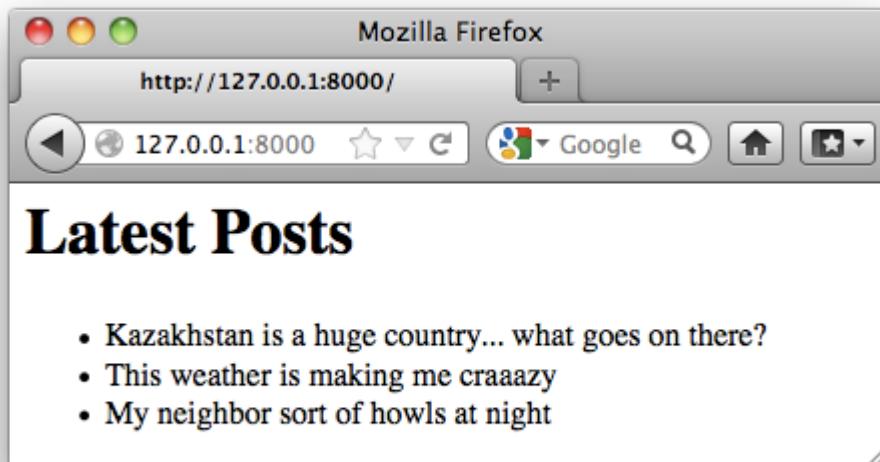


Figure 3.7 Figure 3.1 An HTML response from a web server that queries a database then returns results as a web page.

The JSON file will be formatted like the following snippet, an array of objects with title properties.

```
[ {
  "title": "Kazakhstan is a huge country... what goes on there?"
},
{
  "title": "This weather is making me craaazy"
},
{
  "title": "My neighbour sort of howls at night"
}]
```

How this application works is visually represented by figure 3.2. Note the role of callbacks in the application.

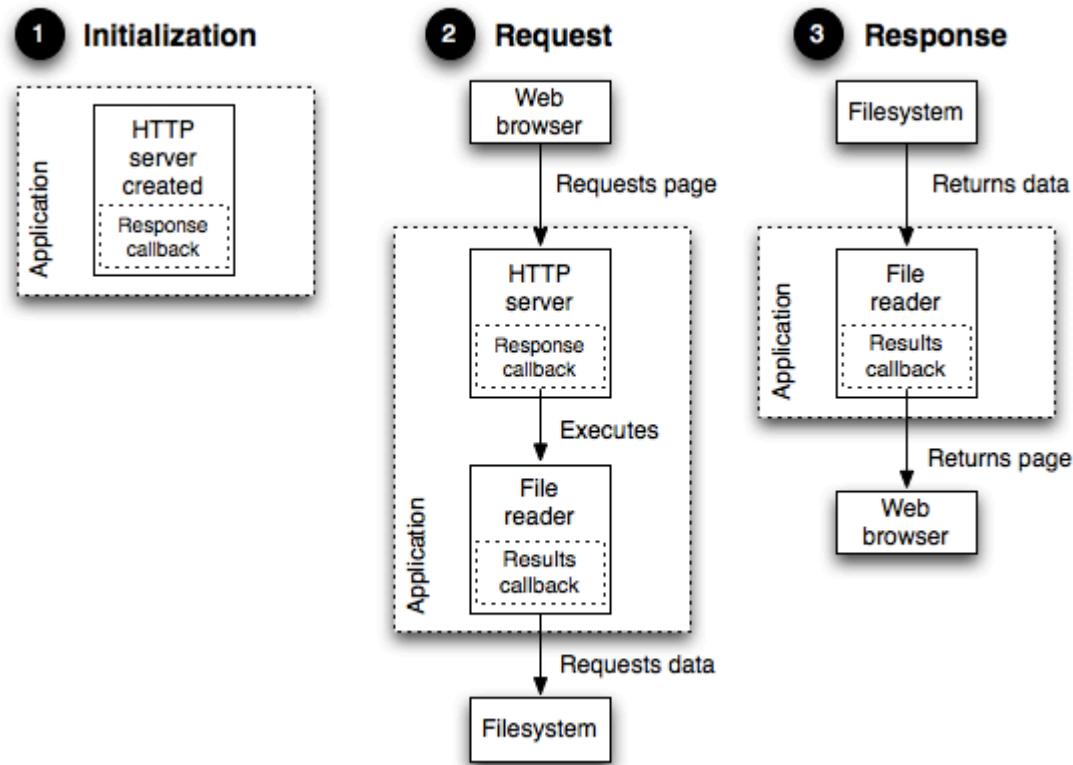


Figure 3.8 How callbacks are used to define response logic in a simple application.

Listing 3.4 contains the code for this example application.

Listing 3.6 blog_recent.js: an example showing the use of callbacks in a simple application

```

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

http.createServer(function(req, res) {
    if (req.url == '/') {
        fs.readFile('../entries.json', function(err, data) {
            if (err) throw err;

            entries = JSON.parse(data.toString());

            var output = '<html><head></head><body>' +
                '<h1>Latest Posts</h1>' +
                '<ul>';
            for (var index in entries) {
                output += '<li>' + entries[index].title + '</li>';
            }
            output += '</ul></body></html>';

            res.writeHead(200, { 'Content-Type': 'text/html' });
            res.end(output);
        });
    }
});

```

```

        });
    }
}).listen(8000, "127.0.0.1");

```

- ➊ Require necessary modules
- ➋ Create an HTTP server and use a callback to define response logic
- ➌ Read text file and use a callback to define what to do with its contents
- ➍ Parse data from JSON text
- ➎ Assemble an HTML page showing the blog titles
- ➏ Send the HTML page to the user

In Node development you may need to create asynchronous functions that require multiple callbacks as arguments, to, for example, handle success or failure. Listing 3.4 shows this idiom. In the example the text "I handle success." is logged to the console.

Listing 3.7 multiple_callbacks.js: an example of the use of multiple callbacks as arguments to a single asynchronous function.

```

function doSomething() {
  return true;
}

function asyncFunction(failure, success) {
  if (doSomething()) {
    success();
  } else {
    failure();
  }
}

asyncFunction(
  function() { console.log('I handle failure.'); },
  function() { console.log('I handle success.'); }
);

```

Node's built-in modules often utilize callbacks with two arguments: one argument for an error, should one occur, and one argument for results. The error argument is often abbreviated as "er" or "err". Here's a typical example of this common function signature.

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

fs.readFile('./some_text_file.txt', function(error, data) {
  if (error) throw error;
  // do something with data if no error has occurred
});
```

Using functions as callback arguments can be messy. The following listing has three levels of nested callbacks. Three levels isn't bad, but the more levels of callbacks you use, the more cluttered your code looks so it's good to try to limit callback nesting.

```
someAsyncFunction('data', function(text1) {
  anotherAsyncFunction(text1, function(text2) {
    yetAnotherAsyncFunction(text2, function(text3) {
      console.log(text3);
    });
  });
});
```

By creating functions that handle the individual levels of callback nesting you can express the same logic in a way that requires more lines of code, but could be considered easier to read, depending on your tastes. Listing 3.5 shows an example of this.

Listing 3.8 reducing_nesting.js: An example of reducing nesting by creating intermediary functions

```
function someAsyncFunction(data, cb) {
  console.log('Executing someAsyncFunction');
  setTimeout(cb, 1000, data);
}

function anotherAsyncFunction(data, cb) {
  console.log('Executing anotherAsyncFunction');
  setTimeout(cb, 1000, data);
}

function yetAnotherAsyncFunction(data, cb) {
  console.log('Executing yetAnotherAsyncFunction');
```

1 Example asynchronous function

2 Example asynchronous function

3 Example asynchronous

```

    setTimeout(cb, 1000, data);
}

function handleResult(text) {
  console.log(text);
}

function innerLogic(text) {
  yetAnotherAsyncFunction(text, handleResult);
}

function outerLogic(text) {
  anotherAsyncFunction(text, innerLogic);
}

someAsyncFunction('some data', outerLogic);

```

function

④ This callback gets called last

⑤ This callback gets called second

⑥ The callback gets called first

⑦ Initiate execution of chain of logic

Now that you've learned how to use callbacks to handle one-off events for such tasks as defining responses when reading text files, web server requests, and reading files, let's move on to how to organize event responses using event emitters.

3.3.2 Handling repeating events with event emitters

Event emitters are entities that manage event listeners and allow events to be triggered. Some important Node API components, such as HTTP servers, TCP/IP servers, and streams, are implemented as event emitters and you can create your own for your own application needs.

As we mentioned earlier, event responses are handled through the use of "listeners." A listener is the association of an event type with an asynchronous callback that gets triggered each time the event type occurs.

For example, if you want logic to act on any data received by a Node server, implemented as an event emitter, it would work as illustrated by figure 3.3.

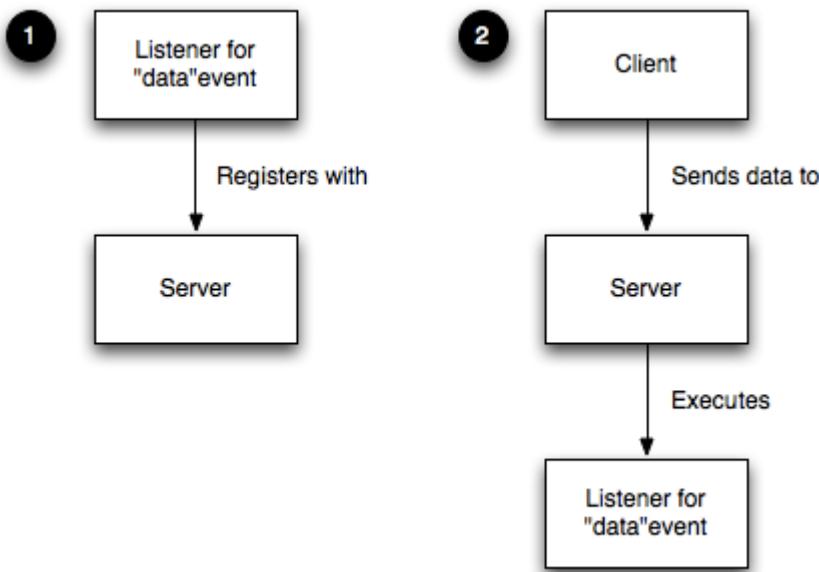


Figure 3.9 Figure 3.3 How an event listener can be used to act on data received by a Node server.

AN EXAMPLE EVENT Emitter

You could use this technique to create an echo server, which, when you sent data to it, will echo the data, as shown in figure 3.4.

```

Last login: Sun Nov 27 15:08:28 on ttys000
Mike-Cantelons-MacBook!~$ telnet 127.0.0.1 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Line one
Line one
Line two
Line two

```

Figure 3.10 An echo server repeating the data sent to it.

Listing 3.6 includes the code needed to implement an echo server. Whenever a client connects a socket is created. The socket is implemented using an event emitter with a listener defined, using the `on` event emitter method, to respond to "data" event types.

Listing 3.9 echo_server.js: An example of the use of an event emitter

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

var server = net.createServer(function(socket) {
  socket.on('data', function(data) {
    socket.write(data);
  });
});

server.listen(8888);
```

You run this echo server by entering the following command:

```
node echo_server.js
```

Once the echo server is running, you can connect to it by entering the following command.

```
telnet 127.0.0.1 8888
```

WARNING

Telnet on Windows

If you're using the Microsoft Windows operating system, telnet may not be installed by default and you'll have to follow instructions⁴ specific to the version of Windows you're running.

Footnote 4

[http://technet.microsoft.com/en-us/library/cc771275\(v=ws.10\).aspx](http://technet.microsoft.com/en-us/library/cc771275(v=ws.10).aspx)

RESPONDING TO A EVENT THAT SHOULD ONLY OCCUR ONCE

Listeners can be defined to repeatedly respond to events, as the previous example showed, or listeners can be defined to respond only once. The following code, using the `once` method, modifies the previous echo server example to only echo the first chunk of data sent to it:

```

var net = require('net');

var server = net.createServer(function(socket) {
  socket.once('data', function(data) {
    socket.write(data);
  });
});

server.listen(8888);

```

CREATING EVENT EMITTERS

In the previous example we used a built-in Node API that leverages event emitters. Node's built-in "events" module, however, allows you to create your own event emitters. The following code defines a "channel" event emitter with a single listener that responds to someone joining the channel. Note that you use `on` (or, alternatively, the longer form `addListener`) to add a listener to an event emitter.

```

var events = require('events');

var channel = new events.EventEmitter();

channel.on('join', function() {
  console.log("Welcome!");
});

```

This code, however, won't do anything when run as there is nothing to trigger an event. You could add a line to the listing that would trigger an event using the `emit` function:

```
channel.emit('join');
```

In chapter 2 you built a chat application that leverages the Socket.io module for

publish/subscribe capabilities. Lets look at how you're implement your own publish/subscribe logic. If you run the script in listing 3.7 you'll have a simple chat server. A chat server channel is implemented as an event emitter that responds to "join" events emitted by clients. When a client joins the channel, the join listened logic, in turn, adds an additional client-specific listener to the channel for the "broadcast" event that will write any message broadcast to the client socket. The names of event types such as "join" and "broadcast" are completely arbitrary. You could use other names for these event types if you wished.

Listing 3.10 pubsub.js: a simple publish/subscribe system using an event emitter.

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

var channel = new events.EventEmitter();
channel.clients = {};
channel.subscriptions = {};

channel.on('join', function(id, client) {
    this.clients[id] = client;
    this.subscriptions[id] = function(senderId, message) {
        if (id != senderId) {
            this.clients[id].write(message);
        }
    }
    this.on('broadcast', this.subscriptions[id]);
});

var server = net.createServer(function (client) {
    var id = client.remoteAddress + ':' + client.remotePort;
    client.on('connect', function() {
        channel.emit('join', id, client);
    });
    client.on('data', function(data) {
        data = data.toString();
        channel.emit('broadcast', id, data);
    });
});
server.listen(8888);
```

- ➊ Add a listener for the "join" event that stores a user's client object, allowing the application to send data back to the user.
- ➋ Ignore data if it's been directly broadcast by the user.
- ➌ Add a listener, specific to the current user, for the "broadcast" event.
- ➍ Emit a "join" event when a user connects to the server, specifying the user ID and

client object.

- ⑤ Emit a channel broadcast event, specifying the user ID and message, when any user sends data.

Once you have the chat server running, open a new command line and enter the following code to enter the chat. If you open up a few command lines you'll see that anything typed in one command line is echoed to the others:

```
telnet 127.0.0.1 8888
```

The problem with this chat server, however, is that when users close their connection and leave the chat room, they leave behind a listener that will attempt to write to a client that is no longer connected. This will, of course, generate an error. To fix this issue, you need to add the listener in listing 3.8 to the channel event emitter and add logic to the server's "close" event listener to emit the channel's "leave" event. The "leave" event, essentially, removes the "broadcast" listener originally added for the client.

Listing 3.11 pubsub.js: Creating a listener to clean up when clients disconnect.

```
...
channel.on('leave', function(id) {
    channel.removeListener(
        ↗'broadcast', this.subscriptions[id]);
    channel.emit('broadcast', id, id + " has left the chat.\n");
});

var server = net.createServer(function (client) {
    ...
    client.on('close', function() {
        channel.emit('leave', id);
    });
});
server.listen(8888);
```

- ➊ Create listener for "leave" event
- ➋ Remove "broadcast" listener for a specific client
- ➌ Emit a "leave" event when client disconnects

If you want to prevent a chat for some reason, but don't want to shut down the server, you could use the `removeAllListeners` event emitter method to remove all listeners of a given type. The following code shows how this could be implemented for our chat server example.

```
channel.on('shutdown', function() {
  channel.emit('broadcast', '', "Chat has shut down.\n");
  channel.removeAllListeners('broadcast');
});
```

You could then add support for a chat command that would trigger the shutdown by changing the listener for the "data" event to the following code:

```
client.on('data', function(data) {
  data = data.toString();
  if (data == "shutdown\r\n") {
    channel.emit('shutdown');
  }
  channel.emit('broadcast', id, data);
});
```

Now when any chat participant enters "shutdown" into chat, it'll cause all chat participants to be kicked off.

SIDE BAR**Error Handling**

A convention to use when creating event emitters is to emit an 'error' type event, providing an error object as an argument, instead of directly throwing an error. This allows custom event response logic to be defined by setting one or more listeners for this event type.

The following code shows how an error listener handles an emitted error by logging into the console.

```
var events = require('events');
var myEmitter = new events.EventEmitter();

myEmitter.on('error', function(err) {
  console.log('ERROR: ' + err.message);
});

myEmitter.emit('error', new Error('Something is wrong.'));
```

If no listener for this event type is defined, however, when the event type 'error' is emitted the event emitter will output a stack trace (a list of program instructions that executed up to the point when the error occurred) and halt execution. The stack trace will indicate an error of the type specified by the emit call's second argument. This behavior is unique to 'error' type events (when other event types are emitted, but have no listeners, nothing happens).

If 'error' is emitted without an error object supplied as the second argument a stack trace will result indicating an "Uncaught, unspecified 'error' event" error and your application will halt. There is a deprecated method you use to deal with this error. You can define your own response by defining a global handler using the following code:

```
process.on('uncaughtException', function(err){
  console.error(err.stack);
  process.exit(1);
});
```

Alternatives to this, such as domains⁵ are being developed, but are considered experimental.

Footnote 5 <http://nodejs.org/api/domain.html>

If you want to provide users connecting to chat with a count of currently connected users you could use the `listeners` method, as shown by the following code, which returns an array of listens for a given event type:

```
channel.on('join', function(id, client) {
  var welcome = "Welcome!\n"
    + 'Guests online: ' + this.listeners('broadcast').length;
  client.write(welcome + "\n");
  ...
});
```

To increase the number of listeners an event emitter has, and to avoid the warnings Node will display once there are more than ten listeners, you could use the `setMaxListeners` method. Using our channel event emitter as an example, you'd use the following code to increase the number of allowed listeners:

```
channel.setMaxListeners(50);
```

EXTENDING THE EVENT Emitter

If you'd like to build upon the event emitter's behavior, you can create a new JavaScript class that inherits from the event emitter. An example of this would be to create a class called "Watcher," which will process files placed in a specified filesystem directory. You'll then use this class to create a utility that watches a filesystem directory (renaming any files placed in it to lower case) and copies the files into a separate directory.

There are three steps to extending an event emitter:

- Creating a class constructor
- Inheriting the event emitter's behavior
- Extending the behavior

The following code shows how you'll create the constructor for our Watcher class. The constructor takes, as arguments, the directory to monitor and the directory in which to put the altered files:

```
function Watcher(watchDir, processedDir) {
  this.watchDir      = watchDir;
  this.processedDir = processedDir;
}
```

Next, you'd add logic to inherit the event emitter's behavior:

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

util.inherits(Watcher, events.EventEmitter);
```

Note the use of the `inherits` function, which is part of Node's built-in `util` module. The `inherits` function provides a clean way to inherit another object's behavior. The `inherits` statement in the previous code snippet is equivalent to the following JavaScript:

```
Watcher.prototype = new events.EventEmitter();
```

After setting up the `Watcher` object, you'd extend the methods inherited from `EventEmitter`, as shown in listing 3.10, with two new methods.

Listing 3.12 event_emitter_extend.js: extending the event emitter's functionality

```
var fs = require('fs')
, watchDir = './watch'
, processedDir = './done';

Watcher.prototype.watch = function() {
  var watcher = this;
  fs.readdir(this.watchDir, function(err, files) {
    if (err) throw err;
    for(var index in files) {
      watcher.emit('process', files[index]);
    }
  })
}
```

1
2

3

```
Watcher.prototype.start = function() {
  var watcher = this;
  fs.watchFile(watchDir, function() {
    watcher.watch();
  });
}
```

4

- ① Extend EventEmitter with method that processes files
- ② Store reference to Watcher object for use in readdir callback
- ③ Process each file in the watch directory
- ④ Extend EventEmitter with method to start watching

The `watch` method cycles through the directory, processing any files found. The `start` method starts the directory monitoring. The monitoring leverages Node's `fs.watchFile` function so when something happens in the watched directory, the `watch` method is triggered, cycling through the watched directory and emitting a 'process' event for each file found.

Now that you've defined the `Watcher` class, you can put it to work by creating a `Watcher` object, using the following code:

```
var watcher = new Watcher(watchDir, processedDir);
```

With your newly created `Watcher` object, you can use the `on` method, inherited from the event emitter class, to set the logic used to process each file, as shown in this snippet:

```
watcher.on('process', function process(file) {
  var watchFile      = this.watchDir + '/' + file;
  var processedFile = this.processedDir + '/' + file.toLowerCase();

  fs.rename(watchFile, processedFile, function(err) {
    if (err) throw err;
  });
});
```

Now that all the necessary logic is in place, you can start the directory monitor using the following code:

```
watcher.start();
```

After putting the Watcher code into a script, and creating "watch" and "done" directories, you should be able to run the script using Node, drop files into the "watch" directory, and see the files pop up, renamed to lower case, in the "done" directory. This is an example of how the event emitter can be a useful class from which to create new classes.

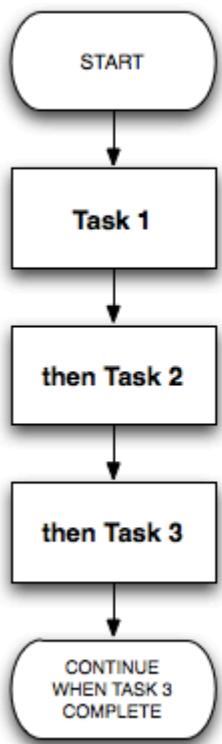
By learning how to use callbacks to define one-off asynchronous logic and how to use event emitters to dispatch asynchronous logic repeatedly, you're one step closer to mastering the control of Node application behavior. In a single callback or event emitter listener, however, you may want to include logic that performs additional asynchronous tasks. If the order in which these tasks are to be performed is important, you may be faced with a new challenge: how to control exactly when each task, in a series of asynchronous tasks, executes.

3.4 Sequencing asynchronous logic

During the execution of an asynchronous program, there are some tasks that can happen any time, independent of what the rest of the program is doing, without causing problems. There are other tasks, however, that should happen only before or after certain tasks.

The concept of sequencing groups of asynchronous tasks is called "flow control" by the Node community. There are two types of flow control: "serial" and "parallel," as figure 3.10 shows.

Serial execution



Parallel execution

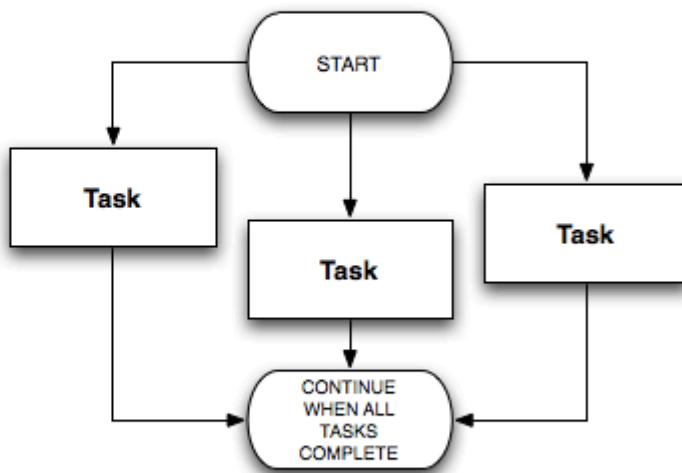


Figure 3.11 Serial execution of asynchronous tasks is similar, conceptually, to synchronous logic: tasks are executed in sequence. Parallel tasks, however, don't have to execute one after another.

Tasks that need to happen one after the other are called "serial." A simple example would be the task of creating a directory then storing a file in it. You wouldn't be able to store the file before creating the directory.

Tasks that don't need to happen one after the other are called "parallel." It isn't necessarily important when these tasks start and stop relative to one another, but they should be all be completed before further logic executes. One example would be downloading a number of files that will later be compressed into a ZIP compressed archive. The files can be downloaded simultaneously, but all of the downloads should be completed before creating the archive.

Keeping track of "serial" and "parallel" involves programmatic "bookkeeping." When you implement "serial" flow control, you need to keep track of the task currently executing or maintain a queue of unexecuted tasks. When you implement "parallel" flow control, you need to keep track of how many tasks have executed to completion.

Flow control tools handle the bookkeeping for you, which makes grouping asynchronous "serial" or "parallel" tasks easy. Although there are plenty of

community created add-ons that deal with sequencing asynchronous logic, implementing flow control yourself demystifies it and helps you gain a deeper sense of how to deal with the challenges of asynchronous programming.

In this section we'll show you how to implement flow control yourself or with community created tools. To start, we're going to look at serial control flow.

3.4.1 When to use serial control flow

In order to execute a number of asynchronous tasks in sequence, you could use callbacks, but if you have a significant number of tasks you'll have to organize them. If you don't, you'll end up with messy code due to excessive callback nesting.

The following code is an example of executing tasks in sequence using callbacks. The example uses `setTimeout` to simulate tasks that take time to execute: the first task takes one second, the next takes half of a second, and the last takes one-tenth of a second. Although the code is short, it's arguably a bit messy and there's no easy way to programmatically add an additional task.

```
setTimeout(function() {
  console.log('I execute first.');
  setTimeout(function() {
    console.log('I execute next.');
    setTimeout(function() {
      console.log('I execute last.');
    }, 100);
  }, 500);
}, 1000);
```

You'll use a flow control tool call `nimble` to execute these tasks. `nimble` is straightforward to use and benefits from having the smallest codebase (a mere 837 bytes, minified and compressed). Install `nimble` with the following command:

```
npm install nimble
```

Now, use the code in listing 3.11 to re-implement the previous code snippet using serial control flow.

Listing 3.13 serial_control_with_tool.js: serial control using a community created

add-on

```
var flow = require('nimble');

flow.series([
  function (callback) {
    setTimeout(function() {
      console.log('I execute first.');
      callback();
    }, 1000);
  },
  function (callback) {
    setTimeout(function() {
      console.log('I execute next.');
      callback();
    }, 500);
  },
  function (callback) {
    setTimeout(function() {
      console.log('I execute last.');
      callback();
    }, 100);
  }
]);

```

1 Provide an array of functions for nimble to execute one after the other

While the implementation using control flow means more lines of code, some would consider it easier to read and maintain. You're likely not going to use control flow all the time, but if you run into a situation where you want to avoid callback nesting, for the sake of code legibility, it's a handy tool.

Now that we've seen an example of the use of serial control flow with a specialized tool, let's look at how to implement it from scratch.

3.4.2 Implementing serial control flow

In order to execute a number of asynchronous tasks in sequence using serial control flow, you first need to put the tasks in an array, in the desired order of execution. This array, as figure 3.7 shows, will act as a queue: when you finish one task you extract the next task in sequence from the array.

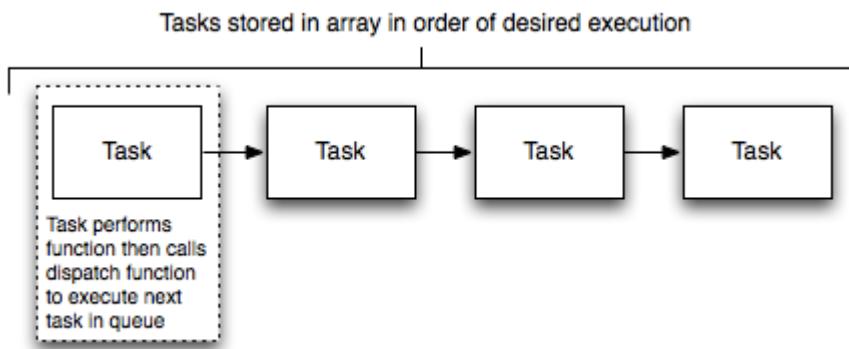


Figure 3.12 Figure 3.7 A visual representation of how serial control flow works.

Each task exists in the array as a function. Once each task has completed, the task should call a handler function to indicate error status and results. The handler function in this implementation will halt execution if there is an error. If there isn't an error, the handler will pull the next task from the queue and execute it.

To show an example of an implementation of serial control flow, we're going to make a simple application that will display a single article's title and URL from a randomly chosen RSS feed. The list of possible RSS feeds will be specified in a text file. The application's output will look something like the following text:

```

Of Course ML Has Monads!
http://lambda-the-ultimate.org/node/4306
  
```

Our example requires the use of two helper modules from the npm repository. First, open a command-line prompt then enter the following commands to create a directory for the example and install the helper modules. The `request` module is a simplified HTTP client that we can use to fetch RSS data with. The `htmlparser` module has functionality that will allow us to turn raw RSS data into JavaScript data structures.

```

mkdir random_story
cd random_story
npm install request
npm install htmlparser
  
```

Next, create a file named `random_story.js` inside this directory that contains the code in listing 3.11:

Listing 3.14 random_story.js: serial control flow implemented in a simple application

```

var fs = require('fs')
, request = require('request')
, htmlparser = require('htmlparser');

var tasks = [
    function() {
        var configFilename = './rss_feeds.txt';
        fs.exists(configFilename, function(exists) {
            if (!exists) {
                next('Create a list of RSS feeds in the file ./rss_feeds.txt.');
            } else {
                next(false, configFilename);
            }
        });
    },
    function (configFilename) {
        fs.readFile(configFilename, function(err, feedList) {
            if (err) {
                next(err.message);
            } else {
                feedList = feedList
                    .toString()
                    .replace(/^\s+|\s+$/g, '')
                    .split("\n");
                var random = Math.floor(Math.random()
                    * feedList.length);
                next(false, feedList[random]);
            }
        });
    },
    function(feedUrl) {
        request({uri: feedUrl}, function(err, response, body) {
            if (err) {
                next(err.message);
            } else if(response.statusCode == 200) {
                next(false, body);
            } else {
                next('Abnormal request status code.');
            }
        });
    },
    function(rss) {
        var handler = new htmlparser.RssHandler();
        var parser = new htmlparser.Parser(handler);
        parser.parseComplete(rss);
    }
];

```

```

if (handler.dom.items.length) {
    var item = handler.dom.items.shift();
    console.log(item.title);
    console.log(item.link);
} else {
    next('No RSS items found.');
}
};

function next(err, result) { ⑨
    if (err) throw new Error(err); ⑩
    var currentTask = tasks.shift(); ⑪
    if (currentTask) { ⑫
        currentTask(result);
    }
}

next(); ⑬

```

- ➊ The application's core functionality is composed of a number of tasks. Each task that needs to be performed by the application is defined as an anonymous function and added to an array.
- ➋ The first task is to make sure the file containing the list of RSS feed URLs, "rss_feeds.txt", exists.
- ➌ The second task starts by reading a file containing the feed URLs.
- ➍ The second task continues by converting the list of feed URLs to a string, removing leading or trailing whitespace for each line, and converting the text list into an array of feed URLs.
- ➎ The second task completes with the selection of a random feed URL from the array of feed URLs.
- ➏ The third task uses the third-party "request" module to do an HTTP request for the randomly selected feed.
- ➐ The fourth task starts by parsing the RSS data from the previous HTTP request into array of items.
- ➑ The fourth task continues by displaying the title and URL of the first feed item, if it exists.
- ➒ A function called "next" handles the execution of each task.
- ➓ If a task encounters an error, an exception is thrown.
- ➔ The next task comes from the array of tasks.
- ➕ Execute the current task.
- ➖ Start the serial execution of tasks.

Before trying out the application, create the file `rss_feeds.txt` in the same directory as the application script. Put the URLs of RSS feeds into the text file, one on each line of the file. Once you've created this file open a command line and

enter the following commands to change to the application directory and execute the script:

```
cd random_story
node random_story.js
```

Serial control flow, as this example implementation shows, is essentially a way of putting callbacks into play when they're needed, rather than simply nesting them.

Now that we know how to implement serial flow control, let's look at how to execute asynchronous tasks in parallel.

3.4.3 Implementing parallel control flow

In order to execute a number of asynchronous tasks in parallel, we again need to put the tasks in an array, but this time the order of the tasks is unimportant. Each task should, once asynchronous logic is complete, call a handler function that will increment the number of completed tasks. Once all tasks are complete, the handler function, when called, should perform some subsequent logic.

To show an example of an implementation of parallel control flow, we're going to make a simple application that will read the contents of a number of text files and output a count of the frequency of word use throughout the files. Reading the contents of the text files will be done using the asynchronous `readFile` function so, conceptually, a number of file reads could be done in parallel. How this application works is shown visually in figure 3.12.

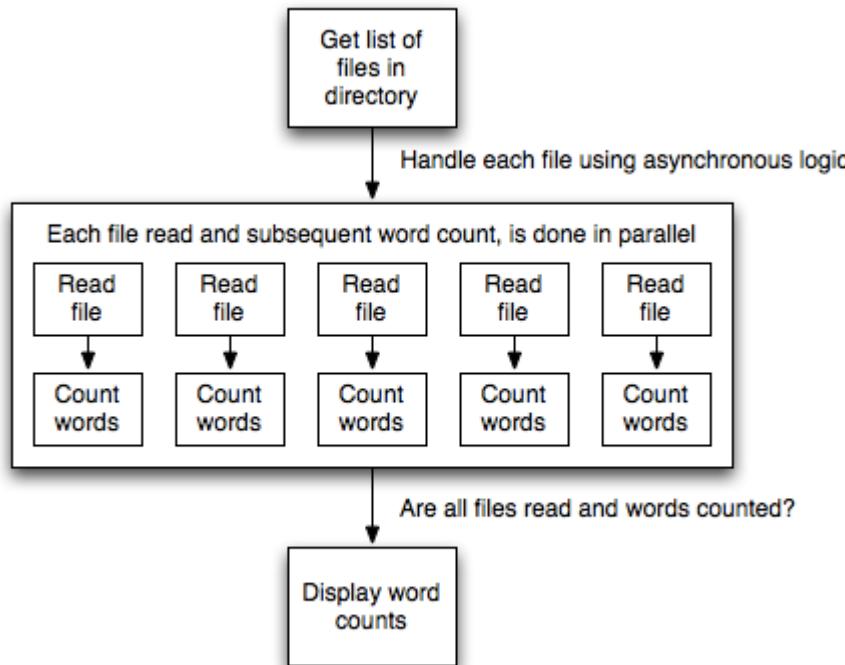


Figure 3.13 Figure 3.8 A visual representation of using parallel control flow to implement a frequency count of word use in a number of files.

The output will look something like the following text (although likely much longer):

```
would: 2
wrench: 3
writeable: 1
you: 24
```

Open a command-line prompt then enter the following commands to create a directory for the example, and a directory within that in which to place text files to analyze:

```
mkdir word_count
cd word_count
mkdir text
```

Next, create a file named `word_count.js` inside this directory that contains

the code in listing 3.12:

Listing 3.15 word_count.js: parallel control flow implemented in a simple application

```

var fs = require('fs')
, completedTasks = 0
, tasks = []
, wordCounts = {}
, filesDir = './text';

function checkIfComplete() {
    completedTasks++;
    if (completedTasks == tasks.length) {
        for(var index in wordCounts) {
            console.log(index +': ' + wordCounts[index]);
        }
    }
}

function countWordsInText(text) {
    var words = text
        .toString()
        .toLowerCase()
        .split(/\W+/)
        .sort();
    for(var index in words) {
        var word = words[index];
        if (word) {
            wordCounts[word] = (wordCounts[word]) ? wordCounts[word] + 1 : 1;
        }
    }
}

fs.readdir(filesDir, function(err, files) {
    if (err) throw err;
    for(var index in files) {
        var task = (function(file) {
            return function() {
                fs.readFile(file, function(err, text) {
                    if (err) throw err;
                    countWordsInText(text);
                    checkIfComplete();
                });
            }
        })(filesDir + '/' + files[index]);
        tasks.push(task);
    }
    for(var task in tasks) {
        tasks[task]();
    }
});

```

1

2

3

4

5

6

- ➊ When all tasks have completed, a list displays of each word used in the files and how many times it was used.
- ➋ Counts word occurrences in text
- ➌ Gets a list of the files in the "text" directory.
- ➍ Defines a task to handle each file. Each task includes a call to a function that will asynchronously read the file then count the file's word usage.
- ➎ Adds each task to an array of functions to call in parallel.
- ➏ Starts executing every task.

Before trying out the application, create some text files in the "text" directory you created earlier. Once you've created these files open a command line and enter the following commands to change to the application directory and execute the script:

```
cd word_count
node word_count.js
```

Now that you've learned how serial and parallel control flow work under the hood, let's look at how to leverage community created tools that allow you to easily benefit from control flow in your applications, without having to implement it yourself.

3.4.4 Leveraging community tools

Many community add-ons exist that provide convenient flow control tools⁶. Some popular add-ons include nimble, step, and seq. Although each of these is worth checking out, we'll use nimble again for another example.

Footnote 6 <http://www.infoq.com/articles/surviving-asynchronous-programming-in-javascript>

Listing 3.13 is an example of using nimble to sequence tasks in a script that downloads two files simultaneously, using parallel flow control, then archives them.

WARNING
The following example won't work in Microsoft Windows

Because the Windows operating system doesn't come with the `tar` and `curl` commands available, the following example won't work in this operating system.

We use serial control to make sure that the downloading is done before proceeding to archiving.

Listing 3.16 control_flow_example.js: the use of a community add-on control flow tool in a simple application

```
var flow = require('nimble')
, exec = require('child_process').exec;

function downloadNodeVersion
  ↗(version, destination, callback) {
  var url = 'http://nodejs.org/dist/node-v' + version + '.tar.gz';
  var filepath = destination + '/' + version + '.tgz';
  exec('curl ' + url + ' >' + filepath, callback);
}

flow.series([
  function (callback) {                                ①
    flow.parallel([
      function (callback) {
        console.log('Downloading Node v0.4.6...');
        downloadNodeVersion('0.4.6', '/tmp', callback);
      },
      function (callback) {
        console.log('Downloading Node v0.4.7...');
        downloadNodeVersion('0.4.7', '/tmp', callback);
      }
    ], callback);
  },
  function(callback) {
    console.log('Creating archive of downloaded files...');
    exec(
      'tar cvf node_distros.tar /tmp/0.4.6.tgz /tmp/0.4.7.tgz',
      function(error, stdout, stderr) {
        console.log('All done!');
        callback();
      }
    );
  }
]);

```

- ① Download Node source code for a given version
- ② Execute a series of tasks in sequence
- ③ Execute downloads in parallel
- ④ Create archive file

The script defines a helper function that will download any specified release version of the Node source code. Two tasks are then executed in series: the parallel

downloading of two versions of Node and the bundling of the downloaded versions into a new archive file.

3.5 Summary

In this chapter, you've learned how to organize your application logic into reusable modules and how to make asynchronous logic behave the way you want it to .

Node's module system, which is based on the CommonJS module specification⁷ , allows you to easily reuse modules by populating the `exports` and `module.exports` objects. The module lookup system affords you a lot of flexibility in terms of where you can put modules and have them be found by application code when you `require` them: in addition to allowing you to include modules in your application's source tree, you can also use the "node_modules" folder to share module code between multiple applications. Within modules, the `package.json` file can be used to specify which file in the module's source tree is first evaluated when the module is required.

Footnote 7 <http://www.commonjs.org/specs/modules/1.0/>

To manage asynchronous logic, you can use callbacks, event emitters, and flow control.

Callbacks are appropriate for one-off asynchronous logic, but their use requires care to prevent messy code. Event emitters can be helpful for organizing asynchronous logic as they allow it to be associated with a conceptual entity and easily managed through the use of "listeners."

The use of flow control allows you to manage how asynchronous tasks execute, either one after another, or simultaneously. Implementing your own flow control is possible, but community add-ons exist that can save you the trouble. Which flow control add-on you prefer is largely a matter of taste and project/design constraints.

Now that you've spent this chapter and the last preparing for development, it's time to sink your teeth into one of Node's most important features: its HTTP APIs. In the next chapter you'll learn the basics of web application development using Node.

Building Node web applications



In this chapter:

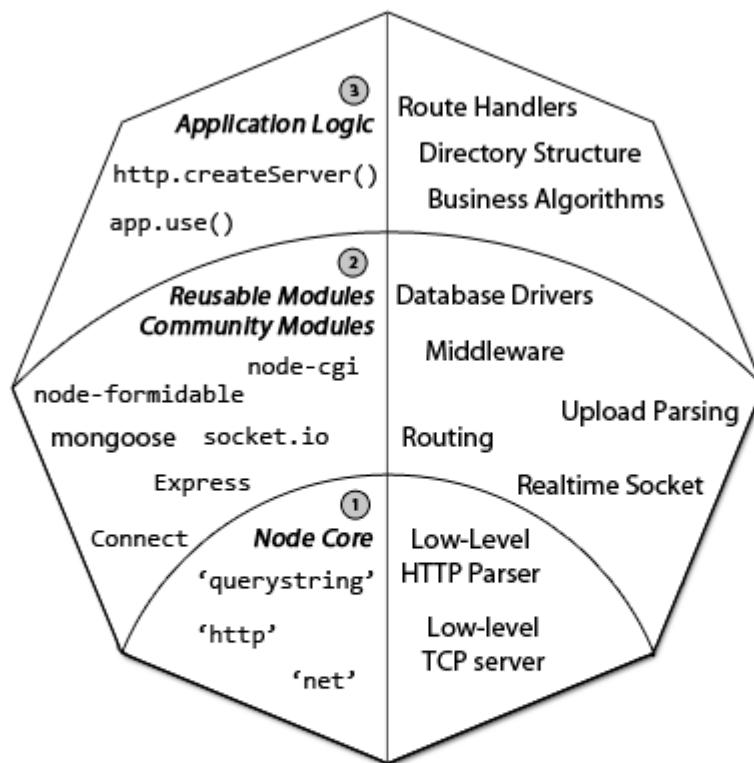
- Handling HTTP requests with Node's API
- Building a RESTful web service
- Serving static files
- Accepting user input from forms
- Securing your application with HTTPS

In this chapter you will become familiar with the tools Node provides us to create HTTP servers, as well as get acquainted with the `fs` (filesystem) module, necessary for serving static files. You will also learn about handling additional common web application needs such as creating an example low-level RESTful web service, accepting user input through an HTML form, monitoring file upload progress, and finally securing your web application with Node's secure socket layer (SSL).

At the core of Node is a powerful streaming HTTP parser consisting of roughly 1,500 lines of optimized C, written by the author of Node, Ryan Dahl. Coupled with the low-level TCP API that Node exposes to JavaScript, you are provided with a very low level, however very flexible HTTP server.

As with the majority of Node's core, the HTTP module favors simplicity. High-level "sugar" APIs are left for 3rd-party frameworks such as Connect or Express that help greatly simplify the web application building process. Figure 4.1 illustrates the anatomy of a Node web application, showing you how the low-level APIs remain at the core, and abstraction and implementations are built on top of those building blocks.

Anatomy of a Node Web Application



1 **Node's Core APIs** are always lightweight and low-level. This leaves opinions, syntactic "sugar" and specific details up to the community modules.

2 **Community modules** are where Node thrives, taking the low-level core APIs and creating fun and easy to use modules to get tasks done easily.

3 The final layer is the **Application Logic**, where "your app" is implemented. The size and this layer depends on the number of community modules utilized, and the complexity of the application.

Figure 4.1 Overview of the layers that make up a Node web application

This chapter will be covering some of Node's low-level APIs directly. You can safely skip this chapter if you are more interested in higher level concepts and web frameworks, like Connect or Express, which will be covered in later chapters. Before creating rich web applications with Node you'll need to become familiar with the fundamental HTTP API, which can be built upon to create higher level tools and frameworks.

4.1 HTTP server fundamentals

Like we've mentioned throughout this book, Node has a relatively low-level API, and the story is no different for HTTP. Node's HTTP interface is lower level than what you will find in familiar frameworks or languages such as PHP, with the ultimate goal of being fast, and flexible.

To begin your mission of creating a robust and performant web application we'll take a look at:

- How Node presents incoming HTTP requests to developers.
- How to write a basic HTTP server that responds with "Hello World".
- How to read incoming request headers and set outgoing response headers.
- How to set the status code of an HTTP response.

Before you can accept incoming requests you need to create an HTTP server! Let's take a look at Node's HTTP interface.

4.1.1 How Node presents incoming HTTP requests to developers

Node provides its HTTP server and client interfaces in the `http` module, so require that in your code first. To create an HTTP server, call the `http.createServer()` function. It accepts a single argument, a callback function, which will be the "request" handler used for each HTTP request received by the server. The callback receives, as arguments, the `req` and `res` objects, which are commonly shortened to `req` and `res`.

```
var http = require('http');
var server = http.createServer(function(req, res){
  // handle request
});
```

For every HTTP request received by the server, the given callback function will be invoked with new `req` and `res` instances, after the HTTP headers have been parsed. This allows Node to accept incoming connections and begin parsing requests. Then you, the developer, can apply application logic.

Typically, someone on the Internet would enter the URL of your website into their browser, the browser then opens a TCP connection to your Node HTTP server, and sends an HTTP request to the server. Node parses *up to the end of the HTTP headers* before invoking the request handler, as illustrated in figure 4.2. This

greatly differs from the approach of PHP, where the entire program is executed in the context of a request. Node servers are long-running processes which will serve many requests throughout its life-time.

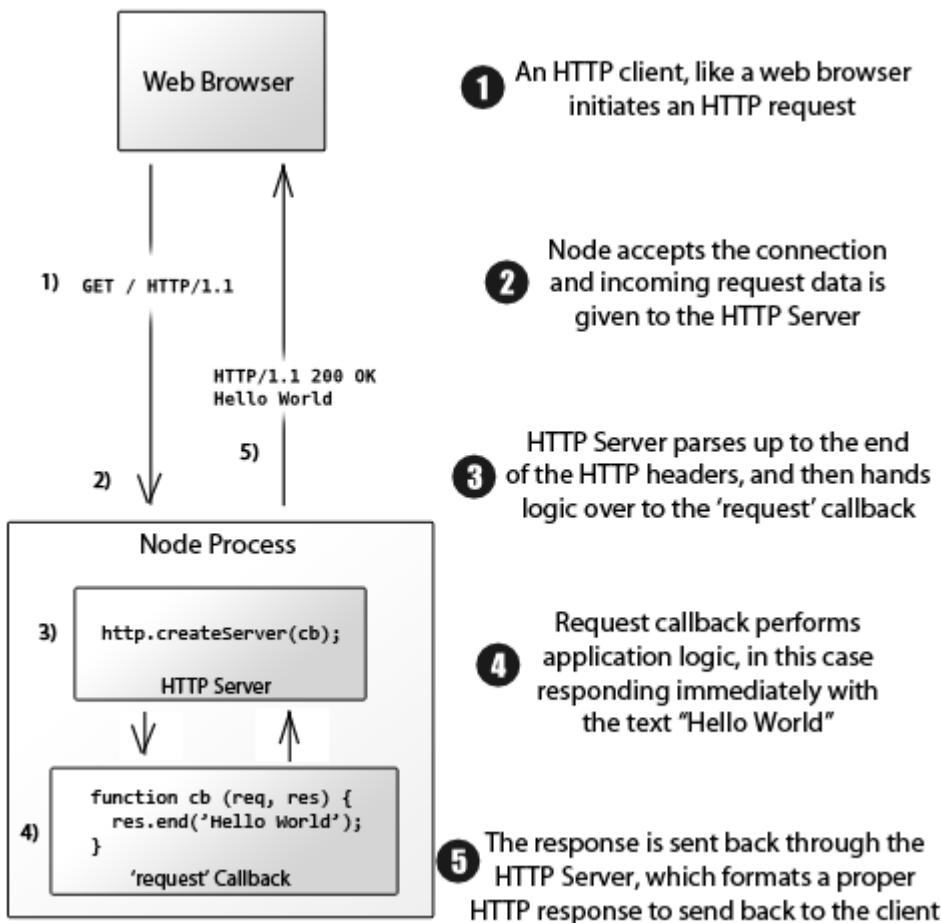


Figure 4.2 The lifecycle of an HTTP request going through a Node HTTP server

4.1.2 A basic HTTP server that responds with "Hello World"

Now let's get on with the fun. To implement the famous "hello world" example, you only have to modify the previous empty server code with 2 additional lines. First, invoke the aptly named `res.write()` method on the response object, writing data to the socket. Then use the `res.end()` method to finish the response.

```
var http = require('http');
var server = http.createServer(function(req, res){
  res.write('Hello World');
  res.end();
```

```
});
```

You now have a fully functional HTTP server! But you have one step remaining, you need to bind the server to a port, and listen for connections. The `server` object returned by `http.createServer()` is an instance of `http.Server`, inheriting from `net.Server` defined in Node's `net` module. The method that is used to listen for connections is named `server.listen()`, and is defined by the `net` module at the TCP level, as the HTTP protocol is built on top of TCP.

This method accepts a combination of arguments, but for now the focus will be on listening for connections with a specified port and IP address. During development it's typical to bind to an unprivileged port such as 3000 on the loopback interface (127.0.0.1, a.k.a. "localhost"), for local use.

```
var http = require('http');
var server = http.createServer(function(req, res){
  res.end('Hello World');
});
server.listen(3000, '127.0.0.1');
```

1 `res.write` and
`res.end` can be
combined

Now that the server is set up for accepting connections and handling requests, you may visit "http://localhost:3000" in your browser, resulting in a plain-text page consisting of the words "Hello World". The use of `res.end('Hello World')` is functionally equivalent to a `res.write('Hello World')` call followed by a call to `res.end()`.

Setting up an HTTP server is just the start, you'll need to set response status codes, header fields, handle exceptions appropriately, as well as get an introduction with higher level concepts using the APIs Node provides. Next you'll take a look in greater detail at responding to incoming requests.

4.1.3 Reading request headers and setting response headers

The "Hello World" example in the previous section is only the bare minimum required for a proper HTTP response. It uses the default status code of 200 (indicating "success") and the default response headers. Usually though, you will want to specify any number of other HTTP headers to include with the response. For example, you will have to send a 'Content-Type' header with a value of 'text/html' when you are sending HTML content as the response of an HTTP request, so that the browser knows to render the result as HTML.

Node offers several methods to progressively alter a HTTP response's header fields. These are the `res.setHeader(field, value)`, `res.getHeader(field)`, and `res.removeHeader(field)` methods. You can add and remove headers in any order, *but* only up to the first `res.write()` or `res.end()` call. After the first part of the response body is written, Node will flush the HTTP headers that have been set.

```
var body = 'Hello World';
res.setHeader('Content-Length', body.length);
res.setHeader('Content-Type', 'text/plain');
res.end(body);
```

The equivalent program in PHP may look similar to the following:

```
<?php
$body = 'Hello World';
header('Content-Length: ' . strlen($body));
header('Content-Type: text/plain');
echo $body;
...
?>
```

4.1.4 Setting the status code of an HTTP response

Additionally, it is common to want to send back a different HTTP status code than the default of 200. A common use-case would be sending back a 404 "Not Found" status code when an HTTP endpoint does not exist. To do this, you set the `res.statusCode` property. This property may be assigned at any point during the application's response, as long as it's before the first call to `res.write()` or `res.end()`. As shown in the following example, this means `res.statusCode = 302` may be placed above the `res.setHeader()` calls, or below.

```
var url = 'http://google.com';
var body = '<p>Redirecting to <a href="' + url + '">' +
           + url + '</a></p>';
res.setHeader('Location', url);
res.setHeader('Content-Length', body.length);
res.setHeader('Content-Type', 'text/html');
res.statusCode = 302;
res.end(body);
```

Node's philosophy is to provide small but robust networking APIs, not to contend with high-level frameworks such as Rails or Django, yet serving as a tremendous platform for similar frameworks to emerge. Because of this design, neither high level concepts like sessions nor fundamentals such as HTTP cookies are provided within core, but rather are left for 3rd-party modules to provide.

Now that you have seen the basic HTTP API, it's time to put them to use. In the next section you will make a simple, HTTP-compliant application using this API.

4.2 Building a RESTful web service

Suppose you want to create a TODO list web service with Node, involving the typical Create, Read, Update, Delete (CRUD) actions. These interactions may be implemented in many ways, however in this section the focus will be on creating a RESTful web service. That is, a service that utilizes the HTTP method verbs to expose a concise API, patterns useful for any web service.

In 2000, the term "REST" or "Representational State Transfer" was introduced by Roy Fielding, one of the prominent contributors to the HTTP 1.0 and 1.1 specifications. By convention, HTTP verbs, such as GET, POST, PUT, DELETE, are mapped to retrieving, creating, updating, and removing the resource(s) specified by the URL. RESTful web services have gained in popularity for being

both simple to utilize and implement in comparison to protocols such as the Simple Object Access Protocol (SOAP).

Throughout this section `curl`¹ will be used in place of a web browser to interact with your web service. `curl` is a powerful command-line HTTP client which can be used to send requests to a target server. To create a compliant REST server, you should implement the four HTTP verbs. Each verb will cover a different task for our TODO list:

Footnote 1 <http://en.wikipedia.org/wiki/CURL>

- The `POST` verb will add items to the TODO list.
- The `GET` verb will display a listing of the current items or display the details of a specific item.
- The `DELETE` verb will remove items from the TODO list.
- Typically, the `PUT` verb should modify existing items, but for brevity's sake we're going to skip `PUT` in this chapter.

4.2.1 Creating resources with `POST` requests

In RESTful terminology, the creation of a "resource" is typically mapped to the `POST` verb. Therefore, the `POST` verb will "create" an entry into the TODO list. In Node, you can check which HTTP method (verb) is being used by checking the `req.method` property as shown in listing 4.1. After knowing which method the request is using, your server will know which task to perform.

As Node's `http-parser` streams data, you receive "chunks" of data in the form of `Buffer` objects, or strings depending on the encoding. By default the "data" events provide `Buffer` objects, which are Node's version of byte arrays. In the case of textual TODO items, there's little interest in binary data, so setting the stream encoding to "ascii" or "utf8" is ideal, as the request will instead emit strings. Listing 4.1 shows how this can be done by invoking the `req.setEncoding(encoding)` method.

In the case of a TODO list item, the entire string is needed before it can be added to the array (also known as "buffering" the response). One way to do this is concatenate all of the chunks until the "end" event is emitted, indicating the request is complete. Once the "end" event has occurred, you will now have the `item` string populated with the entire contents of the request body, which can then be pushed to the `items` array. Now that the item has been added you can end the request with the string "OK" and Node's default status code of 200.

Listing 4.1 todo.js: POST request body string buffering

```

var http = require('http');
var url = require('url');
var items = [];
var server = http.createServer(function(req, res) {
    switch (req.method) { ①
        case 'POST':
            var item = '';
            req.setEncoding('utf8'); ②
            req.on('data', function(chunk){ ③
                item += chunk; ④
            });
            req.on('end', function(){ ⑤
                items.push(item);
                res.end('OK\n');
            });
            break;
    }
}); ⑥

```

- ① The data store is a regular JS Array in memory
- ② req.method is the http method requested
- ③ The string buffer for the incoming item
- ④ Encode incoming data events as utf8 strings
- ⑤ Concatenate the data chunk onto the buffer
- ⑥ After all the data has been received, push the new item onto the items array

This buffering technique shown here may seem confusing at first, but it's a pattern that you will commonly encounter in Node due to its asynchronous nature, so make sure you understand it before moving on. Figure 4.3 illustrates the HTTP server so far handling an incoming HTTP request, buffering the input before acting on the request at the end.

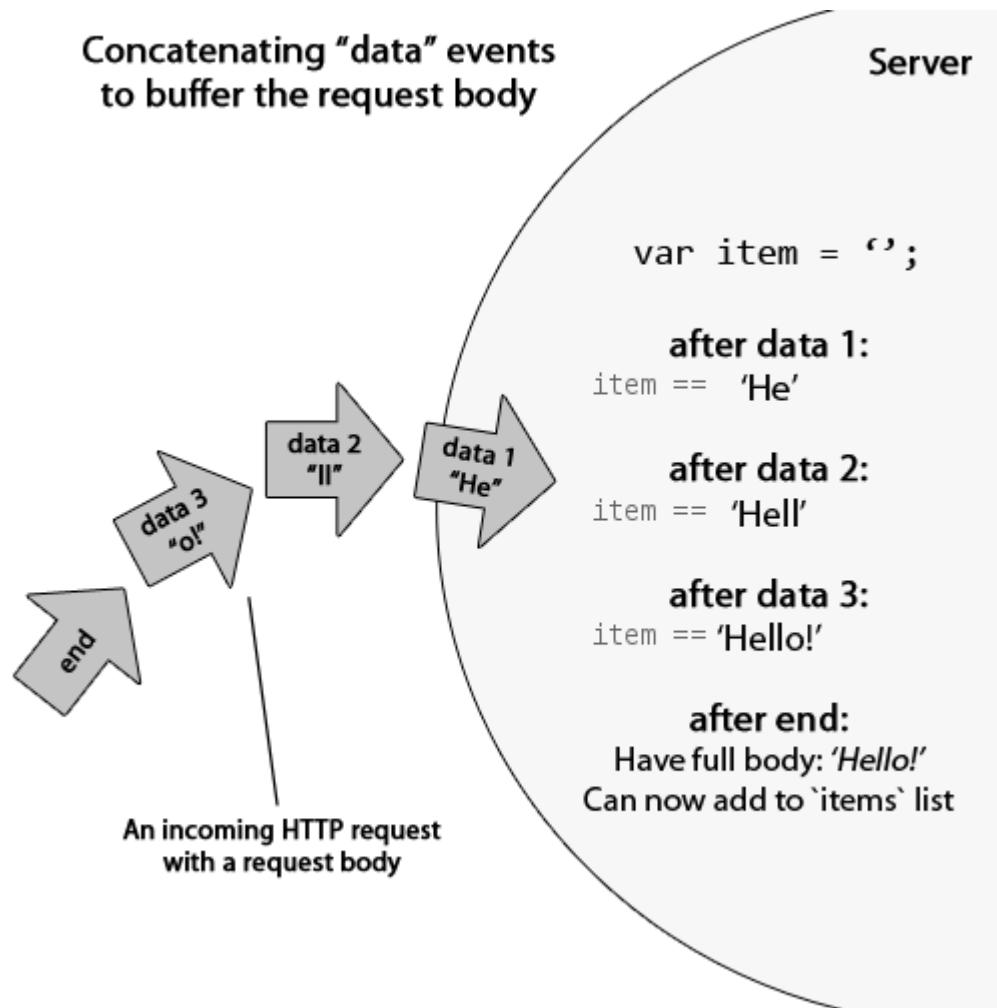


Figure 4.3 Concatenating "data" events to buffer the request body

Now the application can add items, but before trying it out using curl, you should complete the next task so you can get a listing of the items as well.

4.2.2 Fetching resources with *GET* requests

To handle the GET verb add it to the same switch statement as before, followed by the items list logic. In the following example the first call to `res.write()` will write the header with the default fields, as well as the data passed to it.

```
...
case 'GET':
  items.forEach(function(item, i){
    res.write(i + ' ' + item + '\n');
  });
  res.end();
break;
...
```

Now that the app can display the items, it's time to give it a try! Fire up a terminal, start the server, and POST some items using curl's -d flag:

```
$ curl -d 'buy groceries' http://localhost:3000
OK
$ curl -d 'buy node in action' http://localhost:3000
OK
```

Then to GET the list of TODO list items you can execute curl without any flags, as GET is the default verb:

```
$ curl http://localhost:3000
0) buy groceries
1) buy node in action
```

SETTING THE CONTENT-LENGTH HEADER

To speed up responses, when possible the "Content-Length" field should be sent with your response. In the case of the item list, the body can easily be constructed ahead of time in memory, allowing you to access the string length and flush the entire list in one shot. Setting the "Content-Length" header implicitly disables Node's "chunked" encoding, providing a performance boost as less data needs to be transferred. Optimizing the GET handler could look something like:

```
var body = items.map(function(item, i){
  return i + ') ' + item;
}).join('\n');
res.setHeader('Content-Length', Buffer.byteLength(body));
res.setHeader('Content-Type', 'text/plain; charset="utf-8"');
res.end(body);
```

You may be tempted to use the `body.length` value for the "Content-Length", but as the "Content-Length"'s value should represent the byte-length, not character length, this poses a problem if the string contains multi-byte characters. To solve this dilemma, Node gives us the `Buffer.byteLength()` method.

The following Node REPL session using illustrates the issue with using the `string.length` directly, as the 5 character string is comprised of 7 bytes.

```
$ node
> 'etc ...'.length
5
> Buffer.byteLength('etc ...')
7
```

4.2.3 Removing resources with *DELETE* requests

Finally the `DELETE` verb will be used to remove an item. To accomplish this the app will need to check the requested url, which is how the HTTP client will specify which item to remove. In this case, the "identifier" will be the array index in the `items` array, for example `DELETE /1` or `DELETE /5`.

The requested url may be accessed with the `req.url` property, which may contain several components depending on the request. For example if the request was `DELETE /1?api-key=foobar` this property would contain both the pathname and query-string `/1?api-key=foobar`. To parse these meaningful sections Node provides the "url" module, specifically the `.parse()` function. The following node REPL session illustrates the use of this function, splitting up the url into an object, including the `pathname` property you will use in the `DELETE` handler.

```
$ node
> require('url').parse('http://google.com?q=tobi')
{ protocol: 'http:',
  slashes: true,
  host: 'google.com',
  hostname: 'google.com',
  href: 'http://google.com/?q=tobi',
  search: '?q=tobi',
  query: 'q=tobi',
  pathname: '/' }
```

So `url.parse()` parses out only the pathname for you, however the item id is still a string. In order to work with it within the application it should be converted to a number. A simple solution is to use the `String#slice()` method, which returns a portion of the string between two indices. In this case it will be used to skip the first character, giving you just the number portion still as a string. To convert this string to a number it should be passed to the JavaScript global function `parseInt()`, which returns a `Number`.

Listing 4.2 first does a couple checks on the input value, since you can never trust user input to be valid, before responding to the request. If the number is "Not a Number" (the JavaScript value NaN), then the status code is set to 400 indicating a "Bad Request". Following that you check if the item even exists, responding with 404 "Not Found". After the input is validated, then the item may be removed from the `items` array, followed by the app responding with 200 "OK".

Listing 4.2 todo.js: Item DELETE request handler

```
...
case 'DELETE':
  var path = url.parse(req.url).pathname;
  var i = parseInt(path.slice(1), 10);
  if (isNaN(i)) {
    res.statusCode = 400;
    res.end('Invalid item id');
  } else if (!items[i]) {
    res.statusCode = 404;
    res.end('Item not found');
  } else {
    items.splice(i, 1);
    res.end('OK\n');
  }
  break;
...

```

- ➊ Add the `DELETE` case to the switch statement
- ➋ Check that the number is valid
- ➌ Ensure the requested index exists
- ➍ Delete the requested item

You might be thinking 15 lines of code to remove an item from an array is a bit extreme, we promise this is much easier in practice with higher level frameworks providing additional "sugar" APIs. Learning these fundamentals of Node is crucial for understanding, debugging, and enables you to create more powerful applications and frameworks.

A "complete" RESTful service would also implement the PUT HTTP verb, which should modify an existing item in the TODO list. We encourage you to try and implement this final handler yourself, using the techniques found in the code used in this REST server so far, before moving on to the next section, in which you'll learn how to serve static files from your web application.

4.3 Serving static files

Many web applications share similar, if not identical needs, and serving static files (CSS, JavaScript, images) is certainly one of these. While writing a robust and efficient static file server is non-trivial, and robust implementations already exist within Node's community, implementing your own static file server in this section will illustrate Node's low level filesystem API.

In this section you will learn how to:

- Create a simple static file server.
- Optimize the data transfer with `pipe()`.
- Handle user and filesystem errors by setting the status code.

4.3.1 Creating a static file server

Traditional HTTP servers like Apache and IIS act first and foremost as a file server. You might currently have one of these file servers running on some old website, and moving it over to Node, replicating this basic functionality, is an excellent exercise to better understand HTTP servers you have probably used in the past.

Every static file server begins with a "root" directory, which is the starting point from which files requested get served from. In the the server you are going to create, you will define a `root` variable, which will act as the static file server's root directory.

```
var http = require('http');
var parse = require('url').parse;
var join = require('path').join;
var fs = require('fs');
var root = __dirname;
...
```

The "magic" variable `__dirname` is a string defined by Node which is an absolute path to the directory containing your script. So in this case the server will be serving static files relative to the same directory as this script, but you could configure that to any directory path really. The next step is accessing the pathname of the url in order to determine the requested file path.

If the requested path was `"/index.html"`, and our `root` path was `"/var/www/example.com/public"` you can simply join these using the "path"

module's `.join()` method to form the absolute path `"/var/www/example.com/public/index.html"`.

```
var http = require('http');
var parse = require('url').parse;
var join = require('path').join;
var fs = require('fs');
var root = __dirname;
var server = http.createServer(function(req, res){
  var url = parse(req.url);
  var path = join(root, url.pathname);
});
server.listen(3000);
```

SIDE BAR Directory Traversal Attack

Note that this is a simplified file server. If you were wanting to run this in production you should sanitize the input more to avoid users getting access to parts of the file system you did not intend.

Now that you have the path, the contents of the file needs to be transferred. This can be done using high-level streaming disk access with `fs.ReadStream`, one of Node's Stream classes. This class emits "data" events as it incrementally reads our file from the disk. Listing 4.3 implements a simple but fully functional file server.

Listing 4.3 readstream_static.js: Barebones ReadStream static file server

```
var http = require('http');
var parse = require('url').parse;
var join = require('path').join;
var fs = require('fs');
var root = __dirname;
var server = http.createServer(function(req, res){
  var url = parse(req.url);
  var path = join(root, url.pathname);
  var stream = fs.createReadStream(path);
  stream.on('data', function(chunk){
    res.write(chunk);
  });
  stream.on('end', function(){
    res.end();
  });
});
server.listen(3000);
```

1

2

3

4

- ① Construct absolute path
- ② Create an `fs.ReadStream`
- ③ Write file data to the response
- ④ End the response when the file is complete

While this file server would work in most cases, there many more details you will need to consider. Next up you will learn how to optimize the data transfer while making the code for the server even shorter.

OPTIMIZING DATA TRANSFER WITH STREAM.PIPE()

While it's important to know how the `fs.ReadStream` works, and the flexibility its events provide, there's a higher-level mechanism that Node provides for performing the same task. Node aptly names this method `Stream#pipe()`. This method works much a command-line pipe (|) does, one end writes and the other reads. This method allows you to greatly simplify the server code, and its implementation also transparently handles TCP back pressure appropriately.

```
var server = http.createServer(function(req, res){
  var url = parse(req.url);
  var path = join(root, url.pathname);
  var stream = fs.createReadStream(path);
  stream.pipe(res);
});
```

① note: `res.end()` not needed, called internally by `stream.pipe()`

Figure 4.4 shows your HTTP server in the act of reading a static file from the filesystem, and then piping the result to the HTTP client, using `pipe()`.

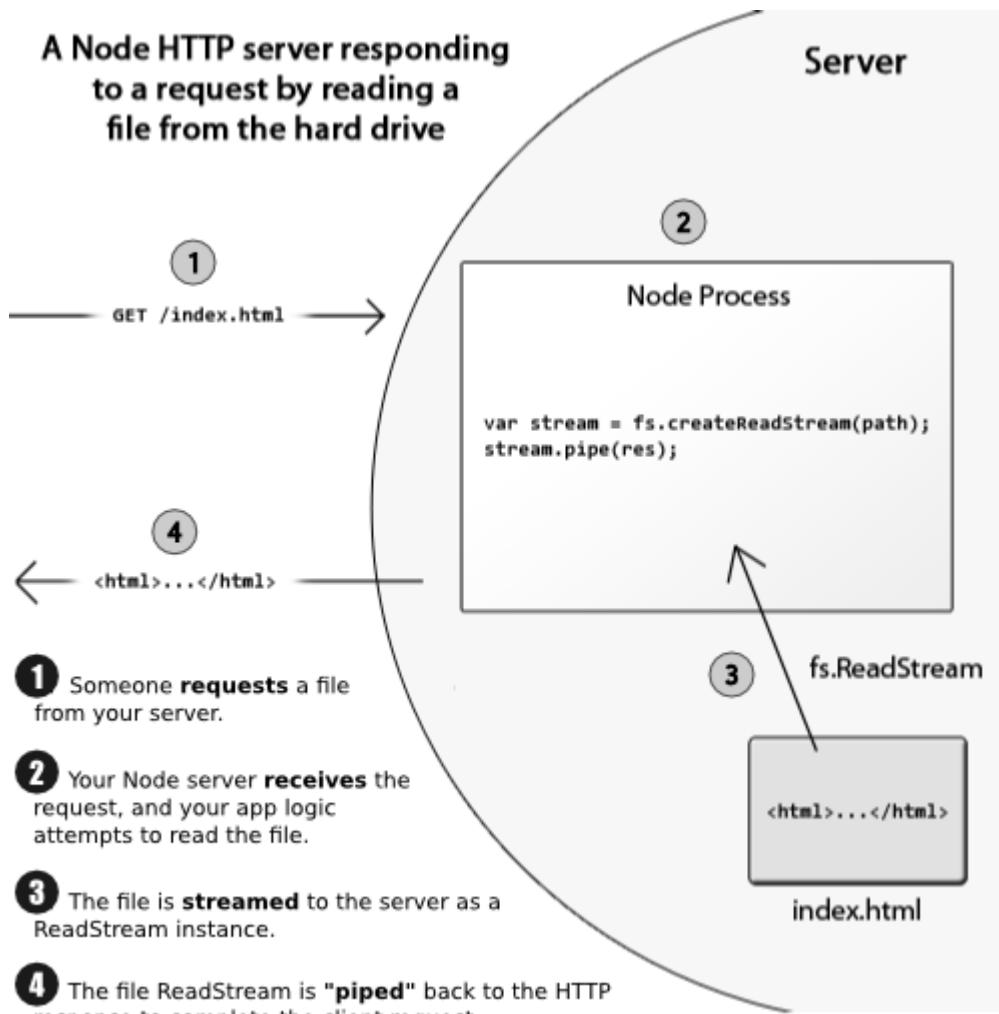


Figure 4.4 A HTTP server serving a static file from the filesystem using `fs.ReadStream`

At this point, you can test to confirm that the static file server is functioning, by executing the following curl command with the `-i` or `--include` flag, instructing curl to output the response header. As previously mentioned, the root directory used is the same directory as the static file server script itself, so in the following curl command you are requesting the server's script itself, which gets sent back as the response body.

```
$ curl http://localhost:3000/static.js -i
HTTP/1.1 200 OK
Connection: keep-alive
Transfer-Encoding: chunked
var http = require('http');
var parse = require('url').parse;
var join = require('path').join;
...
```

This static file server isn't complete yet though, it's still prone to errors, like the user requesting a file that does not exist, and a single unhandled exception will bring down your entire server. In the next section, you will add error handling to the file server.

4.3.2 Handling server errors

Up until now, the static file server has not applied any logic to handle errors emitted from the `fs.ReadStream`. Some examples of when an `error` event would be thrown in the current server are when a file which does not exist is requested, or a request attempted to access a forbidden file, or an I/O related error. In this section we'll touch on how you can make the file server, or any node server more robust.

By default, Node's "error" events throw when no listeners are present. This is true for any `EventEmitter`, even the `fs.ReadStream` instance serving a static file. This is important to know, because *if left unhandled it can take down your entire server*. To illustrate this try requesting a file that does not exist such as `"/notfound.js"`. In the terminal session running your server you'll see the stack trace of an exception printed to `stderr` similar to the following:

```
stream.js:99
throw arguments[1]; // Unhandled 'error' event.
^
Error: ENOENT, No such file or directory
  ' /Users/tj/projects/node-in-action/source/notfound.js '
```

To combat this you will need to register an "error" event handler on the `fs.ReadStream`, which might look something like the following snippet, responding with the 500 response status indicating an internal server error.

```
...
stream.pipe(res);
stream.on('error', function(err){
  res.statusCode = 500;
  res.end('Internal Server Error');
});
...
```

Since the files transferred are indeed static, the `stat()` syscall can be utilized

to request information about a given file, such as the modification time, byte size, and more. These become especially important when providing conditional GET support, where a browser may issue a request in order to check if its cache is stale. Connect's `static()` middleware provides this and many other features such as directory serving, and security related considerations, which we will detail later in chapter 7.

The refactored file server shown in listing 4.4 now implements a call to `fs.stat()`, which responds with an error or an object. When an error has occurred it may be for several reasons, to aid in debugging you can special-case the `err.code` property to respond more directly, for example by responding with 404 "Not Found" for the ENOENT "errno", or error number, which means "No such file or directory", otherwise responding with the generic 500 internal server error status code and message.

Listing 4.4 file_server.js: File server checking for existence and responding with Content-Length

```
var server = http.createServer(function(req, res){
  var url = parse(req.url);
  var path = join(root, url.pathname);
  fs.stat(path, function(err, stat){
    if (err) {
      if ('ENOENT' == err.code) { ①
        res.statusCode = 404;
        res.end('Not Found');
      } else { ②
        res.statusCode = 500;
        res.end('Internal Server Error');
      }
    } else { ③
      res.setHeader('Content-Length', stat.size); ④
      var stream = fs.createReadStream(path);
      stream.pipe(res);
      stream.on('error', function(err){ ⑤
        res.statusCode = 500;
        res.end('Internal Server Error');
      });
    }
  });
}); ⑥
```

- ① parse the url to obtain the pathname
- ② construct absolute path
- ③ check file existence
- ④

- file doesn't exist
- 5 some other error
- 6 set the 'Content-Length' using the stat object

Now that we've taken a low-level look at file serving with Node, let's take a look at an equally common, and perhaps more important feature of web application development, user input from HTML forms.

4.4 Accepting user input from forms

Handling form submissions a very common way of gathering user input for web applications. Node is fairly unique among typical frameworks, as it does not handle the work load for us, we simply get arbitrary body data. This may seem like a bad thing, however like many aspects of Node, this separation of concerns is powerful, leaving opinions to third-party frameworks and providing a fast, robust networking solution.

In this section, we'll take a look at:

- Handling submitted form fields
- Handling uploaded files using node-formidable
- Calculating upload progress in pseudo real-time

4.4.1 Handling submitted form fields

Typically two Content-Type values are associated with form submission requests:

- "application/x-www-form-urlencoded": the default for HTML forms
- "multipart/form-data": used when form contains files, non-ascii, or binary data

In this section, you will re-write the todo list application from the previous section to utilize a form and a web browser. When you're done, you'll have a web-based todo list that looks like the one in the figure below:

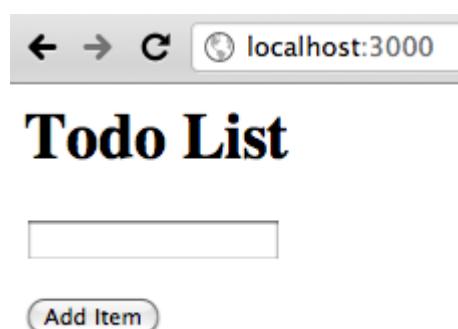


Figure 4.5 A todo-list application utilizing an HTML form and a web browser

To get started, a switch on the request method, or `req.method` is used in listing 4.5 to form simple request routing. Any url that is not *exactly* "/" is considered a 404 "Not Found". Any HTTP verb that is not GET or POST is a 400 "Bad Request". The handler functions `show()`, `add()`, `badRequest()` and `notFound()` will be implemented throughout the rest of this section.

Listing 4.5 http_get_post.js: HTTP server supporting GET and POST

```
var http = require('http');
var items = [];
var server = http.createServer(function(req, res){
  if ('/' == req.url) {
    switch (req.method) {
      case 'GET':
        show(res);
        break;
      case 'POST':
        add(req, res);
        break;
      default:
        badRequest(res);
    }
  } else {
    notFound(res);
  }
});
server.listen(3000);
```

Though typically markup is generated using template engines, for simplicity the example in listing 4.6 just uses string concatenation. There is no need to assign `res.statusCode` because it defaults to 200 "OK". The resulting HTML page looks like the page displayed in figure 4.5 in a web browser.

Listing 4.6 todo_list.js: Todo list form and item list

```
function show(res) {
  var html = '<h1>Todo List</h1>'
  + '<ul>'
  + items.map(function(item){
    return '<li>' + item + '</li>'
  }).join('')
  + '</ul>'
```

① Usually markup is generated using template engines, but for simple apps inlining the HTML works well

```

+ '<form method="post" action="/">'
+ '<p><input type="text" name="item" /></p>'
+ '<p><input type="submit" value="Add Item" /></p>'
+ '</form>';
res.setHeader('Content-Type', 'text/html');
res.setHeader('Content-Length', Buffer.byteLength(html));
res.end(html);
}

```

The `notFound()` function accepts the response object, setting the status code to 404 and response body to "Not Found".

```

function notFound(res) {
  res.statusCode = 404;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Not Found');
}

```

The implementation of the 400 "Bad Request" is nearly identical to `notFound()`, indicating to the client that the request they made was invalid.

```

function badRequest(res) {
  res.statusCode = 400;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Bad Request');
}

```

Saving the most important for last, the application needs to implement the `add()` function which will accept both the `req` and `res` objects. Node has no notion of processing request bodies based on the `Content-Type`. When the form is submitted and the browser issues the request, Node emits "data" events, however it does not parse the body. In this state these "chunks" are nothing more than arbitrary strings, we must check the `Content-Type`, and other influencing header fields such as `Content-Encoding`, and act accordingly.

```

var qs = require('querystring');
function add(req, res) {
  var body = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk){ body += chunk });
  req.on('end', function(){

```

```

    var obj = qs.parse(body);
    items.push(obj.item);
    show(res);
  });
}

```

In the implementation of the `add()` function it's assumed that the Content-Type is "application/x-www-form-urlencoded" for simplicity of this example, so you may simply concatenate the data event chunks to form a complete body string. Since we are not dealing with binary data, you may set the request encoding type to "utf8" with `res.setEncoding()`, signaling Node to decode the data before emitting the "data" event, which now provides strings instead of Buffer objects.

When the request emits the "end" event, all "data" events have completed, and the `body` variable contains the entire body as a string. This works great for small request bodies containing a bit of JSON, XML, etc, however the "buffering" of this data can be problematic, and create an application availability vulnerability if not properly limited to a maximum size, which we will discuss further in the Connect chapter. Because of this it's often beneficial to implement a streaming parser, lowering the memory requirement, and helping to prevent resource starvation. This process incrementally parses the "data" chunks as they are emitted, though this is more difficult to use and implement.

THE QUERYSTRING MODULE

In the server's `add()` function implementation you utilized Node's `querystring` module to parse the body. Let's take a look at a quick REPL session demonstrating how Node's `querystring.parse()` function works, which is the function used in the server. Imagine the user submitted an HTML form to your todo list with the text "take ferrets to the vet":

```

$ node
> var qs = require('querystring');
> var body = 'item=take+ferrets+to+the+vet';
> qs.parse(body);
{ item: 'take ferrets to the vet' }

```

After adding the item, the server returns the user back to the original form again by calling the same `show()` function previously implemented. This is only the

route taken for this example, other approaches could potentially display a message such as "Added todo list item" or redirect the user back to "/".

Try it out! Add a few items and you will see the todo items output in the unordered list. In figure 4.6 you can see the todo list after entering the items "foo", "bar", and "baz" (no doubt a real todo list would have more meaningful items in it). We will not implemented the delete functionality that we did in the REST API previously, but instead leave that as an excercise for you to add.

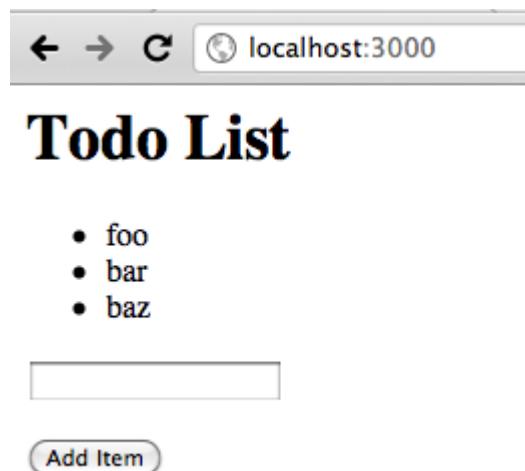


Figure 4.6 The Node todo list application in action

4.4.2 Handling uploaded files using node-formidable

Handling of uploads is another very common, and important aspect of web development. Imagine you are trying to create an application where you upload your photo collection and share it with others using a link on the web. The way to do this using a web browser is through HTML form file uploads.

Below is an example of what a form may look like to upload a file with an associated "name" field.

```
<form method="post" action="/" enctype="multipart/form-data">
<p><input type="text" name="name" /></p>
<p><input type="file" name="file" /></p>
<p><input type="submit" value="Upload" /></p>
</form>
```

To handle file uploads properly and accept the file's content we set the "enctype" attribute to "multipart/form-data", a MIME type better suited for large

BLOBs (Binary Large Objects).

Parsing multipart requests in a performant and streaming fashion is a non-trivial task, and is out of scope for this book; however Node's community has provided several modules to fulfill this need. One such module, "node-formidable", was created by Felix Geisendorfer for his media upload and transformation startup "Transloadit", where performance and reliability are key.

What makes node-formidable a great choice is it's a non-buffering streaming parser, meaning it can accept chunks of data as they arrive, parse them, and emit part-specifics such as the part headers and bodies previously mentioned. Not only is this approach fast, at roughly 500 megabytes per second, the lack of buffering prevents memory bloat, even for very large files such as videos, which otherwise could overwhelm a process.

Now, back to our photo sharing example. The following HTTP server in listing 4.7 implements the beginnings of the file upload server, responding to GET with an HTML form, and an empty function for POST, in which node-formidable will be integrated to handle file uploading.

Listing 4.7 http_server_setup.js: HTTP server setup prepared to accept file uploads

```
var http = require('http');
var server = http.createServer(function(req, res){
  switch (req.method) {
    case 'GET':
      show(req, res);
      break;
    case 'POST':
      upload(req, res);
      break;
  }
});

function show(req, res) {
  var html =
    + '<form method="post" action="/" enctype="multipart/form-data">'
    + '<p><input type="text" name="name" /></p>'
    + '<p><input type="file" name="file" /></p>'
    + '<p><input type="submit" value="Upload" /></p>'
    + '</form>';
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('Content-Length', Buffer.byteLength(html));
  res.end(html);
}

function upload(req, res) {
```

①

```
// upload logic
}
```

➊ Serve an HTML form with a file input

Now that the GET request case is taken care of, it's time to implement the `upload()` function, which is invoked by the request callback when a POST request comes in. The `upload()` function needs to accept the incoming upload data, which is where "node-formidable" comes in. Throughout the rest of this section you will learn what is needed in order to integrate "node-formidable" into your web application:

- Install "node-formidable" through npm.
- Create an `IncomingForm` instance.
- Call `form.parse()` with the HTTP request object.
- Listen for form events "field", "file" and "end".
- Use formidable's "high-level" API

The first step to utilizing "formidable" in the project, is of course, to install it! This can be done by executing the following command, installing the module locally, into the `./node_modules` directory.

```
$ npm install formidable
```

To access the API you should then `require()` it along with the initial `http` module:

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

The first step to implementing the `upload()` function is to respond with 400 "Bad Request" when the request does not appear to be the appropriate type of content.

```
function upload(req, res) {
  if (!isFormData(req)) {
    res.statusCode = 400;
    res.end('Bad Request: expecting multipart/form-data');
```

```

        return;
    }
}

function isFormData(req) {
    var type = req.headers['content-type'] || '';
    return 0 == type.indexOf('multipart/form-data');
}

```

The helper function `isFormData()` checks the "Content-Type" header field for "multipart/form-data", by asserting that it's at the beginning of the field's value via the JavaScript `String` method `indexOf()`.

Now that the request intention is verified, you should initialize a new `formidable.IncomingForm`, followed by the method call `form.parse(req)`, where `req` is the request object, so `formidable` can access the request's "data" events for parsing.

```

function upload(req, res) {
    if (!isFormData(req)) {
        res.statusCode = 400;
        res.end('Bad Request');
        return;
    }
    var form = new formidable.IncomingForm();
    form.parse(req);
}

```

This `IncomingForm` object emits many events itself, and by default, streams file uploads to the "/tmp" directory. As shown in the listing 4.8, `formidable` gives you access to events like "field", and "file", along with common events like "end".

Listing 4.8 formidable.js: Formidable streaming form parser API example

```

...
var form = new formidable.IncomingForm();
form.on('field', function(field, value){
    console.log(field);
    console.log(value);
});
form.on('file', function(name, file){
    console.log(name);
    console.log(file);
});
form.on('end', function(){
    res.end('upload complete!');
})

```

```
});  
form.parse(req);  
...
```

By examining the first two `console.log()` calls in the "field" event handler, you can see that "my clock" was entered in the "name" text field:

```
name  
my clock
```

The "file" event is emitted when the file upload is complete. The `file` object provides us with the file size, its path in the `form.uploadDir` directory ("`/tmp`" by default), original basename, and MIME type. The `file` object looks like the following when passed to `console.log()`:

```
{ size: 28638,  
  path: '/tmp/d870ede4d01507a68427a3364204cdf3',  
  name: 'clock.png',  
  type: 'image/png',  
  lastModifiedDate: Sun, 05 Jun 2011 02:32:10 GMT,  
  length: [Getter],  
  filename: [Getter],  
  mime: [Getter],  
  ...  
}
```

Formidable also provides a higher-level API, essentially wrapping the API we've already looked at into a single callback. When a function is passed to `form.parse()` an `error` is passed as the first argument if something goes wrong, otherwise two objects are passed `fields`, and `files`. The `fields` object may look something like the following `console.log()` output:

```
{ name: 'my clock' }
```

The `files` object provides the same `File` instances that the "file" event emits, keyed by name similarly to `fields`. It's important to note that you may listen on these events even while using the callback, so aspects like progress

reporting are not hindered. The following shows how this more concise API can be used to produce the same results that we've discussed:

```
var form = new formidable.IncomingForm();
form.parse(req, function(err, fields, files){
  console.log(fields);
  console.log(files);
  res.end('upload complete!');
});
```

Now that you have the basics down, we will look at calculating upload progress, a process which comes quite natural to Node and its event loop.

4.4.3 Calculating upload progress in pseudo real-time

Formidable's "progress" event emits the bytes received, and bytes expected. When combined with logic to map uploads you will have a fully functional upload progress bar. In the following example the percentage is computed, and logged, by invoking `console.log()` each time the event is fired.

```
form.on('progress', function(bytesReceived, bytesExpected){
  var percent = Math.floor(bytesReceived / bytesExpected * 100);
  console.log(percent);
});
```

The result of this script, will yield similar to following:

```
1
2
4
5
6
8
...
99
100
```

Now that you understand this concept, the next obvious step would be to relay that progress back to the user's browser. This is a fantastic feature for any

application expecting large uploads, and is a task well suited for Node. A realtime module like socket.io would make it possible in just a few lines of code, but we'll leave that as an exercise for you to figure out.

So now you have the basis of a progress reporting library built, a task that is both more difficult and awkward with many other platforms, letting Node's evented architecture shine. We have one final, and very important topic to cover, securing your application with TLS, Transport Layer Security.

4.5 Securing your application with HTTPS

A frequent requirement for e-commerce sites, and sites dealing with sensitive data, is the need to keep traffic to and from the server private. Standard HTTP sessions involve the client and server exchanging information using unencrypted text. This makes HTTP traffic fairly trivial to eavesdrop on.

The Hypertext Transfer Protocol Secure (HTTPS) protocol provides a way to keep web sessions private. HTTPS combines HTTP with the TLS transport layer. Data sent using TLS is encrypted, and therefore harder to eavesdrop on.

If you'd like to take advantage of HTTPS in your Node application, the first step is generating a private key and a certificate. The private key is, essentially, a "secret" needed to decrypt data sent between the server and client. The private key is kept in a file on the server in a place where it cannot be easily accessed by untrusted users. In this section you will generate what is called a "self-signed certificate". These kinds of SSL certificates cannot be used in production websites because browsers will display a warning message when a page is accessed with a "untrusted" cert, but it is useful for development and testing encrypted traffic.

To generate a private key you will need OpenSSL, which will already be installed on your system if you installed Node. To generate a private key, which we'll call "key.pem", open up a command-line prompt and enter the following:

```
openssl genrsa 1024 > key.pem
```

In addition to a private key, you'll need a certificate. Unlike a private key, a certificate can be shared with the world, and contains a "public key" and information about the certificate holder. The public key is used to encrypt traffic sent from the client to the server. Enter the following to generate a certificate, which we'll call "key-cert.pem", using our private key:

```
openssl req -x509 -new -key key.pem > key-cert.pem
```

Now that you've generated your keys, put them in a safe place. In our example HTTPS server in listing 4.9 we'll reference keys stored in the same directory as our server script, although keys are more often kept elsewhere, typically `~/.ssh`. The following code will create a simple HTTPS server using your keys:

Listing 4.9 https_server.js: HTTPS server options

```
var https = require('https');
var fs = require('fs');
var options = {
  key: fs.readFileSync('./key.pem'),
  cert: fs.readFileSync('./key-cert.pem')
};

https.createServer(options, function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}).listen(3000);
```

- ① The SSL key and cert are provided
- ② by your certificate vendor
- ③ The 'options' object is passed in first
- ④ The https module provides an almost
- ⑤ identical API as the http module

Once that HTTPS server code is running, you can connect to it securely using a web browser. To connect to it, navigate to `https://localhost:3000/` in your web browser. As the certificate used in our example isn't backed by a Certificate Authority, a warning will be displayed. Ignore this warning: it's shown because the public key isn't verifiable by a Certificate Authority (CA), and is considered "untrusted". If you're deploying a public site you should always properly register with a CA² and get a real, trusted SSL certificate for use with your server.

Footnote 2 http://wikipedia.org/wiki/Certificate_authority

4.6 Summary

In this chapter, we've introduced the fundamentals of Node's HTTP server, teaching you how to respond to incoming requests, and how to handle asynchronous exceptions in order to keep your application reliable. You've learned how to create a RESTful web application, serve static files, and even create a pseudo real-time upload progress calculator.

You may also have seen that starting with Node from a web application developer point of view can seem daunting. As seasoned web developers, we promise it's worth the effort, as this knowledge will aid in your understanding of Node for debugging, authoring open-source frameworks, or contributing to existing frameworks.

This fundamental knowledge will prepare you for diving into Connect, a higher-level "framework" providing a fantastic set of bundled functionality that every web application framework can take advantage of. Then there's Express--the icing on the cake! Together these tools will make everything you've learned in this chapter easier, more secure, and more enjoyable.

Before we get there we'll need something to store our application data! In the next chapter we'll look at the rich selection of database clients created by the Node community, which will help power the applications we create throughout the rest of the book.

A large, light gray, stylized number '5' is positioned in the upper right corner of the page.

Storing Node application data

In this chapter

- In-memory and filesystem data storage
- Conventional relational database storage
- Non-relational database storage

Almost every application, web-based or otherwise, requires data storage of some kind—and applications you build with Node will be no different. The choice of an appropriate storage mechanism depends on five factors:

- what data is being stored
- how quickly data needs to be read and written to maintain adequate performance
- how much data exists
- how data needs to be queried
- how long and reliably the data needs to be stored

Methods of storing data range from keeping data in server memory to interfacing with a full-blown database management system (DBMS), but all methods require trade-offs of one sort or another.

Some data is relatively simple (log entries, for example), but some data is more complex, involving different types and the relations between them. Mechanisms that support long-term persistence of complex structured data along with powerful search facilities incur a significant performance cost, so are not always the best strategy. Storing data in server memory, for example, maximizes performance, but is less reliably persistent because data will be lost if the application restarts or the server loses power.

So how will you decide which storage mechanism to use in your applications? In the world of Node application development, it isn't unusual to use different storage mechanisms for different use cases. In this chapter we'll talk about how to store data without having to install and configure a DBMS, and how to store data directly using MySQL, Postgres, Redis, and MongoDB. You'll use some of these storage mechanisms to build applications later in the book and by the end of this chapter you'll know how to use these storage mechanisms to address your own application needs.

To start, let's look at the easiest and lowest level of storage possible: serverless data storage.

5.1 Serverless data storage

From the standpoint of system administration, the most convenient storage mechanisms are those that don't require you to maintain a DBMS to use them, such as in-memory storage and file-based storage. By removing the need to install and configure a DBMS, serverless data storage makes the applications you build much easier to install.

The lack of a DBMS makes serverless data storage a perfect fit for applications you intend others to run on their own hardware. In addition to web applications and other TCP/IP applications, for example, Node can be used to create command-line interface (CLI) tools. A Node-driven CLI tool might require storage, but it's likely the user won't want to go through the hassle of setting up a MySQL server in order to use the tool.

In the next section you'll learn when and how to use in-memory storage and file-based storage, both of which are primary forms of serverless data storage. Let's start with the simplest of the two: in-memory storage.

5.1.1 In-memory storage

In the example applications in chapter 2 and 4, in-memory storage was used to keep track of, respectively, details about chat users and tasks. In-memory storage uses variables to store data. Reading and writing this data is fast, but as we mentioned earlier, we lose the data during server and application restarts.

The ideal use of in-memory storage is for small bits of frequently accessed data. One such application could be for a counter that keeps track of the number of page views since the last application restart. The following code demonstrates the use of in-memory storage for this. Running this code will start a web server on port 8888 that counts each request:

```

var http = require('http')
, counter = 0;

var server = http.createServer(function(req, res) {
  counter++;
  res.write('I have been accessed ' + counter + ' times.');
  res.end();
}).listen(8888);

```

For applications that need to store information that can persist beyond application and server restarts, file-based storage may be more suitable.

5.1.2 File-based storage

File-based storage uses a filesystem to store data. Developers often use it to store application configuration information, but it's also an easy way to persist data to survive application and server restarts.

To illustrate the use of file-based storage, let's create a simple command-line variant of chapter 4's web-based Node application for keeping track of tasks that need to be done.

Figure 5.1 A command-line tool for storing tasks so you don't forget them

The application will store tasks in a file named “.tasks” in whatever directory in which the script runs. Tasks will be converted to JSON before being stored and will be converted from JSON when read from the file.

To create the application, you'll:

- Write the starting logic
- Define a helper function to retrieve tasks
- Define a helper function to store tasks

WRITE THE STARTING LOGIC

The logic begins by requiring the necessary modules, parsing the task command and description from the command-line arguments, and specifying the file in which tasks should be stored, as shown in the following code:

```

var fs = require('fs')
, path = require('path')
, args = process.argv.splice(2)
, command = args.shift()

```

```
, taskDescription = args.join(' ')
, file = path.join(process.cwd(), './tasks');
```

If we provide a command, the application either outputs a list of stored tasks or adds a task description to the task store, as shown in listing 5.1. If we don't provide a command, the command usage will display.

Listing 5.1 cli_tasks.js: Determining what action the CLI script should take

```
switch(command) {
  case 'list':
    listTasks(file);
    break;

  case 'add':
    addTask(file, taskDescription);
    break;

  default:
    console.log('Usage: ' + process.argv[0]
      + ' list|add [taskDescription]');
}
```

DEFINE A HELPER FUNCTION TO RETRIEVE TASKS

The next step will be to define a helper function called `loadOrInitializeTaskArray` in the application logic to retrieve existing tasks. As listing 5.2 shows, `loadOrInitializeTaskArray` loads in a text file in which JSON-encoded data is stored. Two asynchronous `fs` module functions are used in the code.

Listing 5.2 cli_tasks.js: Loading JSON-encoded data from a text file

```
function loadOrInitializeTaskArray(file, cb) {
  fs.exists(file, function(exists) {
    var tasks = [];
    if (exists) {
      fs.readFile(file, 'utf8', function(err, data) {
        if (err) throw err;
        tasks = JSON.parse(data);
      });
    }
    cb(null, tasks);
  });
}
```

1 Check if todo file already exists

2 Read todo data from todo file

```
        var data = data.toString();
        var tasks = JSON.parse(data);
        cb(tasks);
    });
} else {
    cb([]);
}
});
```

- 3 Parse JSON-encoded todo data into an array of tasks
- 4 Create empty array of tasks if file containing todo data doesn't exist

Next, use the `loadOrInitializeTaskArray` helper function to implement the `listTasks` functionality, as shown in the following code:

```
function listTasks(file) {
    loadOrInitializeTaskArray(file, function(tasks) {
        for(var i in tasks) {
            console.log(tasks[i]);
        }
    });
}
```

DEFINE A HELPER FUNCTION TO STORE TASKS

Now you need to define another helper function, `storeTasks`, in the application logic to store JSON-serialized tasks into a file.

```
function storeTasks(file, tasks) {
    fs.writeFile(file, JSON.stringify(tasks), 'utf8', function(err) {
        if (err) throw err;
        console.log('Saved.');
    });
}
```

Then, use the `storeTasks` helper function to implement the `addTask` functionality.

```

function addTask(file, taskDescription) {
  loadOrInitializeTaskArray(file, function(tasks) {
    tasks.push(taskDescription);
    storeTasks(file, tasks);
  });
}

```

Using the filesystem as a datastore makes it relatively quick and easy to add persistence to an application. It's also a great way to handle application configuration. If application configuration data is stored in a text file and encoded in JSON the logic defined earlier in `loadOrInitializeTaskArray` could be repurposed to read the file and parse the JSON.

In chapter 12 (“Beyond web servers”) you’ll learn more about manipulating the filesystem with Node. Now, let’s move on to look at the traditional data storage workhorses of applications: relational database management systems.

5.2 Relational database management systems

RDBMS’s allow complex information to be stored and easily queried. RDBMS’s have traditionally been used for relatively high-end applications such as content management systems, customer relationship management, and shopping carts. They can perform well when used correctly, but require specialized administration knowledge and access to a database server. They also require knowledge of structured query language (SQL), although object-relational mappers (ORMs) exist that provide APIs to write SQL for you in the background. RDBMS administration, ORMs, and SQL are out of the scope of this book, but you’ll find many online resources that cover these technologies.

Developers have many relational database options but most choose open source databases, primarily because they’re well-supported, work well, and don’t cost anything. In this section we’ll look at MySQL and Postgres, the two most popular fully featured relational databases. MySQL and Postgres have similar capabilities and both are solid choices. If you haven’t used either, MySQL is easier to set up and has a larger user base. If you happen to use the proprietary Oracle database, you’ll want to use the `db-oracle`¹ module, which is also outside the scope of this book.

Footnote 1 <https://github.com/mariano/node-db-oracle>

Let’s start with MySQL, then look at Postgres.

5.2.1 MySQL

MySQL is the world's most popular SQL database and is well-supported by the Node community. If you're new to MySQL and interested in learning it, you'll find the official tutorial online². For those new to SQL, many online tutorials and books, including Chris Fehily's *SQL: Visual QuickStart Guide*, are available to help you get up to speed.

Footnote 2 <http://dev.mysql.com/doc/refman/5.0/en/tutorial.html>

USING MYSQL TO BUILD A WORK-TRACKING APP

To see how Node takes advantage of MySQL let's look at an application scenario requiring an RDBMS. Let's say you're creating a serverless web application to keep track of how you spend your workdays. You'd need to keep track of the date of the work, time spent on the work, and a description of the work performed. You could build this web application using the filesystem as a simple datastore, but it would be tricky to build reports with the data. If you wanted to create a report on the work you did last week, for example, you'd have to read every work record stored and check the record's date. Having application data in an RDBMS gives you the ability to generate reports easily using SQL queries.

To build a work tracking application, you could complete the following steps:

- Create the application logic
- Create helper functions you need to make the application work
- Write the functions that let you add, delete, update, and retrieve data with MySQL
- Write the code that renders the HTML records and forms

The end result, as shown in figure 5.2, will be a simple web application that allows you to record work performed, review work recently performed, and archive work recently performed.

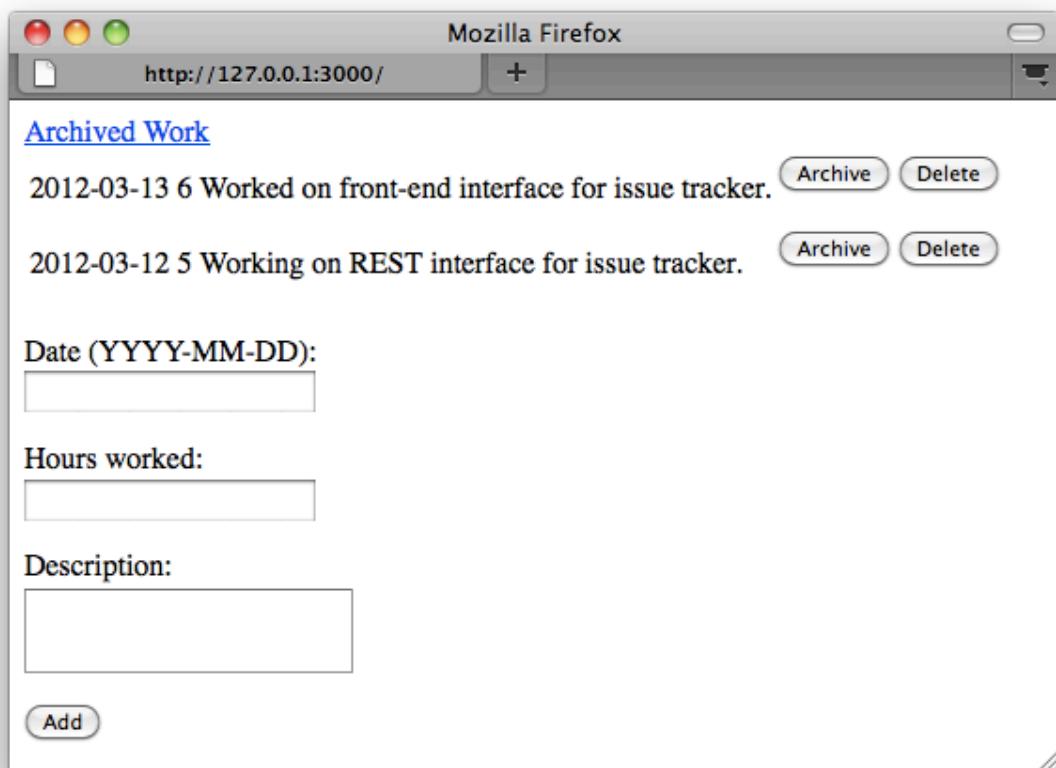


Figure 5.2 A simple web application that allows you to track work performed

To allow Node to talk to MySQL we'll need to use a community-created module that interfaces with MySQL. The most popular MySQL API module is Felix Geisendorfer's `node-mysql`³ module, which we'll use in this section. To begin, install the MySQL Node module using the following command:

Footnote 3 <https://github.com/felixge/node-mysql>

```
npm install mysql
```

CREATING THE APPLICATION LOGIC

Next we need to create two files for application logic. Our application will be composed of two files: `timetrack_server.js`, used to start the application, and `timetrack.js`, a module containing application-related functionality. To start, create a file named `timetrack_server.js` and include the following code in it. This code includes Node's HTTP API, application-specific logic, and an MySQL API. Fill in the `host`, `user`, and `password` settings with those that correspond to your MySQL configuration.

```
var http = require('http')
, work = require('./lib/timetrack')
, mysql = require('mysql');

var db = mysql.createConnection({
  host:      '127.0.0.1',
  user:      'myuser',
  password: 'mypassword',
  database: 'timetrack'
});
```

Next, add the logic in listing 5.3 to define the basic web application behavior. The application allows you to browse, add, and delete work performance records. In addition, the app will let you archive work records. Archiving a work record hides it on the main page. But archived records remain browsable on a separate web page.

Listing 5.3 timetrack_server.js: HTTP request routing

```
var server = http.createServer(function(req, res) {
  switch (req.method) {
    case 'POST':
      switch(req.url) {
        case '/':
          work.add(db, req, res);
          break;
        case '/archive':
          work.archive(db, req, res);
          break;
        case '/delete':
```

1 Route HTTP POST requests

```

        work.delete(db, req, res);
        break;
    }
    break;
  case 'GET':
    switch(req.url) {
      case '/':
        work.show(db, res);
        break;
      case '/archived':
        work.showArchived(db, res);
    }
    break;
  }
);

```

2 Route HTTP GET requests

The code in listing 5.4 is the final addition to `timetrack_server.js`. This logic creates a database table if none exists, and starts the HTTP server listening to IP address 127.0.0.1 on TCP/IP port 3000. All node-mysql queries are performed using the `query` function.

Listing 5.4 timetrack_server.js: Database table creation

```

db.query(
  "CREATE TABLE IF NOT EXISTS work (
  + "id INT(10) NOT NULL AUTO_INCREMENT, "
  + "hours DECIMAL(5,2) DEFAULT 0, "
  + "date DATE, "
  + "archived INT(1) DEFAULT 0, "
  + "description LONGTEXT, "
  + "PRIMARY KEY(id))",
  function(err) {
    if (err) throw err;
    console.log('Server started...');
    server.listen(3000, '127.0.0.1');
  }
);

```

1 Table creation SQL

2 Start HTTP server

CREATING HELPER FUNCTIONS THAT SEND HTML, CREATE FORMS, AND RECEIVE FORM DATA

Now that you've fully defined the file you'll use to start the application, it's time to create the file that defines the rest of the application's functionality. Create a directory named "lib" and inside this directory create a file named `timetrack.js`. Inside this file insert the logic (see listing 5.5), which includes the Node `querystring` API and defines helper functions for sending web page HTML and receiving data submitted through forms.

Listing 5.5 timetrack.js: Helper functions for sending HTML, creating forms, and receiving form data

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

exports.sendHtml = function(res, html) {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('Content-Length', Buffer.byteLength(html));
  res.end(html);
}

exports.parseReceivedData = function(req, cb) {
  var body = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk){ body += chunk });
  req.on('end', function() {
    var data = qs.parse(body);
    cb(data);
  });
}

exports.actionForm = function(id, path, label) {
  var html = '<form method="POST" action="' + path + '">' +
    '<input type="hidden" name="id" value="' + id + '">' +
    '<input type="submit" value="' + label + '" />' +
    '</form>';
  return html;
}
```

➊ Send HTML response

➋ Parse HTTP POST data

➌ Render simple form

ADDING DATA WITH MYSQL

Next, add the following code (see listing 5.6) to `timetrack.js` to define logic that'll add a work record to the MySQL database.

You use the character "?" as a placeholder to indicate where a parameter should

be placed. Each parameter is automatically escaped by the `query` method before being added to the query, preventing SQL injection attacks. Note that the second argument of the `query` method is now a list of values to substitute with the placeholders.

Listing 5.6 timetrack.js: Adding a work record

```
exports.add = function(db, req, res) {
  exports.parseReceivedData(req, function(work) {
    db.query(
      "INSERT INTO work (hours, date, description) " +
      " VALUES (?, ?, ?)",
      [work.hours, work.date, work.description],
      function(err) {
        if (err) throw err;
        exports.show(db, res);
      }
    );
  });
}
```

- ➊ Parse HTTP POST data
- ➋ SQL to add work record
- ➌ Work record data
- ➍ Show user a list of work records

DELETING MYSQL DATA

Next, add the following code (see listing 5.7) to `timetrack.js` to define logic that will delete a work record:

Listing 5.7 timetrack.js: Deleting a work record

```
exports.delete = function(db, req, res) {
  exports.parseReceivedData(req, function(work) {
    db.query(
      "DELETE FROM work WHERE id=?",
      [work.id],
      function(err) {
        if (err) throw err;
        exports.show(db, res);
      }
    );
  });
}
```

- ➊
- ➋
- ➌
- ➍

- 1 Parse HTTP POST data
- 2 SQL to delete work record
- 3 Work record ID
- 4 Show user a list of work records

UPDATING MYSQL DATA

Next, add the following logic (see listing 5.8) to `timetrack.js` to define logic that will update a work record to flag it as “archived.”

Listing 5.8 timetrack.js: Archiving a work record

```
exports.archive = function(db, req, res) {
  exports.parseReceivedData(req, function(work) {
    db.query(
      "UPDATE work SET archived=1 WHERE id=?",
      [work.id],
      function(err) {
        if (err) throw err;
        exports.show(db, res);
      }
    );
  });
}
```

- 1
- 2
- 3
- 4

- 1 Parse HTTP POST data
- 2 SQL to update work record
- 3 Work record ID
- 4 Show user a list of work records

RETRIEVING MYSQL DATA

Now that you’ve defined the logic that’ll add, delete, and update a work record, you’ll next add the logic in listing 5.9 to retrieve work record data—archived or unarchived—so it can be rendered as HTML. When issuing a query that will return the data, the callback that defines what to do after the query should have a “rows” argument, as shown in listing 5.9.

Listing 5.9 timetrack.js: Retrieving work records

```
exports.show = function(db, res, showArchived) {
```

```

var query = "SELECT * FROM work " +
    "WHERE archived=? " +
    "ORDER BY date DESC";
var archiveValue = (showArchived) ? 1 : 0;
db.query(
    query,
    [archiveValue],
    function(err, rows) {
        if (err) throw err;
        html = (showArchived)
            ? ''
            : '<a href="/archived">Archived Work</a><br/>';
        html += exports.workHitlistHtml(rows);
        html += exports.workFormHtml();
        exports.sendHtml(res, html);
    }
);
}

exports.showArchived = function(db, res) {
    exports.show(db, res, true);
}

```

- 1 SQL to fetch work records
- 2 Desired work record archive status
- 3 Format results as HTML table
- 4 Send HTML response to user
- 5 Show only archived work records

RENDERING THE MYSQL RECORDS

Add the logic in listing 5.10 to `timetrack.js` to do the rendering of work records to HTML:

Listing 5.10 timetrack.js: Rendering work records to an HTML table

```

exports.workHitlistHtml = function(rows) {
    var html = '<table>';
    for(var i in rows) {
        html += '<tr>';
        html += '<td>' + rows[i].date + '</td>';
        html += '<td>' + rows[i].hours + '</td>';
        html += '<td>' + rows[i].description + '</td>';
        if (!rows[i].archived) {
            html += '<td>' + exports.workArchiveForm(rows[i].id) + '</td>';
        }
        html += '<td>' + exports.workDeleteForm(rows[i].id) + '</td>';
        html += '</tr>';
    }
}

```

```

    }
    html += '</table>';
    return html;
}

```

- ➊ Render each work record as an HTML table row
- ➋ Show archive button if work record isn't already archived

RENDERING THE HTML FORMS

Finally, add the code in listing 5.11 to `timetrack.js` to render HTML forms needed by the application.

Listing 5.11 timetrack.js: HTML forms for adding, archiving, and deleting work records.

```

exports.workFormHtml = function() {
  var html = '<form method="POST" action="/">' +
    '<p>Date (YYYY-MM-DD):<br/><input name="date" type="text"><p/>' +
    '<p>Hours worked:<br/><input name="hours" type="text"><p/>' +
    '<p>Description:<br/>' +
    '<textarea name="description"></textarea></p>' +
    '<input type="submit" value="Add" />' +
    '</form>';
  return html;
}

exports.workArchiveForm = function(id) {                                1
  return exports.actionForm(id, '/archive', 'Archive');
}

exports.workDeleteForm = function(id) {                                    2
  return exports.actionForm(id, '/delete', 'Delete');
}

```

- ➊ Render blank HTML form for entry of new work record
- ➋ Render archive button form
- ➌ Render delete button form

TRYING IT OUT

Now that you've fully defined the application, you can run it. Make sure that you've created a database named "timetracker" using your MySQL administration interface of choice. Then start the application by entering the following into your command line:

```
node timetrack_server.js
```

Finally, navigate to `http://127.0.0.1:3000/` in a web browser to use the application.

MySQL may be the most popular relational database, but Postgres is, for many, the more respected of the two. Let's look at how to use Postgres in your application.

5.2.2 Postgres

Postgres is well regarded for its standards compliance and robustness, and many Node developers favor it over other RDBMS's. If you're new to Postgres and interested in learning it, you'll find the official tutorial online⁴.

Footnote 4 <http://www.postgresql.org/docs/7.4/static/tutorial.html>

The most mature and actively developed Postgres API module is Brian Carlson's `node-postgres`⁵.

Footnote 5 <https://github.com/brianc/node-Postgres>

WARNING

Untested for Windows

While the `node-postgres` module is intended to work for Windows, the module's creator primarily tests using Linux and OS X, so the Windows users may encounter issues.

Install `node-postgres` via npm using the following command:

```
npm install pg
```

CONNECTING TO POSTGRES

Once you've installed the node-postgres module, you can connect to Postgres, and select a database to query using the following code (omitting the “:mypassword” portion of the connection string if no password is set):

```
var pg = require('pg');
var conString = "tcp://myuser:mypassword@localhost:5432/mydatabase";

var client = new pg.Client(conString);
client.connect();
```

INSERTING A ROW INTO A DATABASE TABLE

The `query` method performs mysql-postgres queries. The following example code shows how to insert a row into a database table:

```
client.query(
  'INSERT INTO users ' +
  '(name) VALUES (''Mike'')'
);
```

Placeholders (“\$1,” “\$2,” and so on) indicate where to place a parameter. Each parameter is escaped before being added to the query, preventing SQL injection attacks. The following example shows the insertion of a row using placeholders:

```
client.query(
  "INSERT INTO users " +
  "(name, age) VALUES ($1, $2)",
  ['Mike', 39]
);
```

To get the primary key value of a row after an insert you can use a

RETURNING clause to specify the name of the column whose value you'd like to return. You'd then add a callback as the last argument of the query call, as the following example shows:

```
client.query(
  "INSERT INTO users " +
  "(name, age) VALUES ($1, $2) " +
  "RETURNING id",
  ['Mike', 39],
  function(err, result) {
    if (err) throw err;
    console.log('Insert ID is ' + result.rows[0].id);
  }
);
```

CREATING A QUERY THAT RETURNS RESULTS

If you're creating a query that will return results, you'll need to store the client query method's return value to a variable. The query method returns an object that has inherited EventEmitter behavior to take advantage of built-in Node functionality. This object emits a 'row' event for each retrieved database row. Listing 5.12 shows how you can output data from each row returned by a query. Note the use of EventEmitter listeners that define what to do with database table rows and what to do when data retrieval is complete.

Listing 5.12 Selecting rows from a PostgreSQL database

```
var query = client.query(
  "SELECT * FROM users WHERE age > $1",
  [40]
);

query.on('row', function(row) {
  console.log(row.name)
});

query.on('end', function() {
  client.end();
});
```

1 Handle return of a row

2 Handle query completion

An ‘end’ event is emitted after the last row is fetched and may be used to close the database or continue with further application logic.

Relational databases may be classic workhorses, but another breed of database manager that doesn’t require the use of SQL is becoming increasingly popular.

5.3 NoSQL databases

In the early days of the database world, non-relational databases were the norm. But relational databases were slowly gaining in popularity and over time became the mainstream choice for applications both on and off the web. In recent years, a resurgent interest in non-relational DBMS has emerged as its proponents claimed advantages in scalability and simplicity. Many of these DBMS’s target a variety of usage scenarios. They are popularly referred to as “NoSQL” databases, interpreted as “No SQL,” or “Not Only SQL.”

In this section we’ll look at two popular NoSQL databases: Redis and MongoDB. We’ll also look at Mongoose, a popular API that abstracts access to MongoDB, adding a number of time-saving features. The setup and administration of Redis and MongoDB are out of the scope of this book, but you’ll find quickstart instructions for Redis⁶ and MongoDB⁷ on the web that should help you get up and running.

Footnote 6 <http://redis.io/topics/quickstart>

Footnote 7 <http://docs.mongodb.org/manual/installation/#installation-guides>

5.3.1 Redis

Redis is a datastore well-suited to handling simple, relatively ephemeral data such as logs, votes, and messages. It provides a vocabulary of primitive but useful commands⁸ that work on a number of data structures. Most of the data structures supported by Redis will be familiar to developers as they are analogous to those frequently used in programming: hash tables, lists, and key/value pairs (which are used like simple variables). Redis also supports a less familiar data structure called a “set,” which we’ll talk about later in this chapter.

Footnote 8 <http://redis.io/commands>

We won’t go into all of Redis’s commands in this section, but we’ll run through a number of examples that’ll be useful for many applications. If you’re new to Redis and want to get an idea of its power before trying these examples, a great place to start is Simon Willison’s “Redis tutorial⁹.”

Footnote 9 <http://simonwillison.net/static/2010/redis-tutorial/>

The most mature and actively developed Redis API module is Matt Ranney's `node_redis`¹⁰ module. Install this module using the following npm command:

Footnote 10 https://github.com/mranney/node_redis

```
npm install redis
```

CONNECTING TO A REDIS SERVER

The following code establishes a connection to a Redis server using the default TCP/IP port running on the same host. The Redis client you've created has inherited `EventEmitter` behavior that emits an "error" event when the client has problems communicating with the Redis server. As the following example shows, you can define your own error-handling logic by adding a listener for the "error" event type:

```
var redis = require('redis'),
    client = redis.createClient(6379, '127.0.0.1');

client.on('error', function (err) {
    console.log('Error ' + err);
});
```

MANIPULATING DATA IN REDIS

After you've connected to Redis, your application can start manipulating data immediately using the `client` object. The following example code shows the storage and retrieval of a key/value pair. The `redis.print` function simply prints the results of an operation or, should an error occur, the error.

```
client.set('color', 'red', redis.print);
client.get('color', function(err, value) {
```

```

    if (err) throw err;
    console.log('Got: ' + value);
});

```

STORING AND RETRIEVING VALUES USING A HASH TABLE

Listing 5.13 shows the storage and retrieval of values in a slightly more complicated data structure: the hash table also referred to as "hash". The `hmset` Redis command sets hash elements, identified by a key, to a value. The `hkeys` Redis command lists the keys of each element in a hash.

Listing 5.13 redis_keys.js: Storing data in elements of a Redis hash

```

client.hmset('camping', {
  'shelter': '2-person tent',
  'cooking': 'campstove'
}, redis.print); 1

client.hget('camping', 'cooking', function(err, value) { 2
  if (err) throw err;
  console.log('Will be cooking with: ' + value);
});

client.hkeys('camping', function(err, keys) { 3
  if (err) throw err;
  keys.forEach(function(key, i) {
    console.log('  ' + key);
  });
});

```

- ① Set hash elements
- ② Get “cooking” element’s value
- ③ Get hash keys

STORING AND RETRIEVING DATA USING THE LIST

Another data structure supported is the list. A Redis list can theoretically hold over four billion elements, memory permitting. The following code shows the storage and retrieval of values in a list. The `lpush` Redis command adds a value to a list. The `lrange` Redis command retrieves a range of list items using a start and end argument. The “-1” end argument in the following code signifies the last item of the list, hence this use of `lrange` will retrieve all list items:

```
client.lpush('tasks', 'Paint the bikeshed red.', redis.print);
client.lpush('tasks', 'Paint the bikeshed green.', redis.print);
client.lrange('tasks', 0, -1, function(err, items) {
  if (err) throw err;
  items.forEach(function(item, i) {
    console.log(' ' + item);
  });
});
```

Redis lists, which are similar conceptually to arrays in many programming languages, provide a familiar way to manipulate data. But one downside to them is their retrieval performance. As a Redis list grows in length, retrieval becomes slower ($O(n)$ in big O notation).

NOTE

Big O notation

In computer science, big O notation is a way of categorizing algorithms by complexity. Seeing an algorithm’s description in big O notation gives a quick idea of the performance ramifications of the algorithm’s use. For those new to big O, Rob Bell’s “Beginner’s Guide to Big O Notation”¹¹ provides a great overview.

Footnote 11

<http://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/>

STORING AND RETRIEVING DATA USING SETS

Sets are another type of Redis data structure, but with better retrieval performance. The time it takes to retrieve a set member is independent of the size of the set ($O(1)$ in big O notation). Sets must contain unique elements -- if you try to store two identical values in a set, the second attempt to store it will be ignored. The following code illustrates the storage and retrieval of IP addresses. The `sadd` Redis command attempts to add a value to the set and the `smembers` command returns stored values. In the example we've twice attempted to add the IP "204.10.37.96," but as you can see, when we display the set members, the IP has only been stored once.

```
client.sadd('ip_addresses', '204.10.37.96', redis.print);
client.sadd('ip_addresses', '204.10.37.96', redis.print);
client.sadd('ip_addresses', '72.32.231.8', redis.print);
client.smembers('ip_addresses', function(err, members) {
  if (err) throw err;
  console.log(members);
});
```

DELIVERING DATA WITH CHANNELS

Redis, it's also worth noting, goes beyond the traditional role of datastore by providing "channels." Channels are data-delivery mechanisms, as shown conceptually in figure 5.3, that provide "publish/subscribe" functionality, useful for chat and gaming applications.

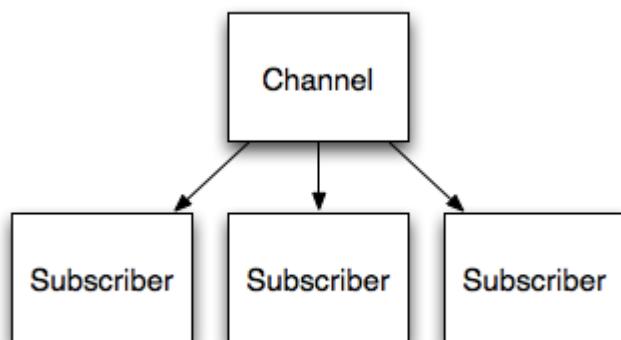


Figure 5.3 Redis channels provide an easy solution to a frequent data delivery scenario

A Redis client can both subscribe to and publish to any given channel. Subscribing to a channel means you get any message sent by others to the channel. Publishing a message to a channel sends the message to everyone else on that channel. Listing 5.14 shows an example of how Redis' publish/subscribe functionality can be used to relay messages.

Listing 5.14 redis_pubsub.js: An example of Redis' publish/subscribe functionality

```
var redis = require('redis')
, clientA = redis.createClient()
, clientB = redis.createClient();

clientA.on('message', function(channel, message) {
  console.log('Client A got message from channel %s: %s',
    channel,
    message
  );
});

clientA.on('subscribe', function(channel, count) {
  clientB.publish('main_chat_room', 'Hello world!');
});

clientA.subscribe('main_chat_room');
```

- ① Create two Redis clients
- ② Define what should happen when a message is received by client A
- ③ Define what should happen once client A is subscribed
- ④ Client B publishes a message to the channel

MAXIMIZING NODE_REDIS PERFORMANCE

When you're deploying a Node.js application to production that uses the node-redis API, you may want to consider using Pieter Noordhuis's hiredis module¹². Using this module will speed up Redis performance significantly because it takes advantage of the official hiredis C library. The API node-redis will automatically use hiredis, if installed, instead of its JavaScript implementation. Install hiredis using the following npm command:

Footnote 12 <https://github.com/pietern/hiredis-node>

```
npm install hiredis
```

Note that because the hiredis library compiles from C code, and Node's internal APIs change occasionally, you may have to recompile hiredis when upgrading Node.js. Rebuild hiredis using the following npm command:

```
npm rebuild hiredis
```

Now that we've looked at Redis, which excels at high-performance handling of data primitives, let's look at a more generally useful database: MongoDB.

5.3.2 MongoDB

MongoDB is a general-purpose non-relational database. It's used for the same sorts of applications for which you would use an RDBMS.

A MongoDB database stores documents in "collections." Documents in a collection, as shown in figure 5.4, need not share the same schema: each document could conceivably have a different schema. This makes MongoDB more flexible than conventional RDBMS's, as you don't have to worry about predefining schema.

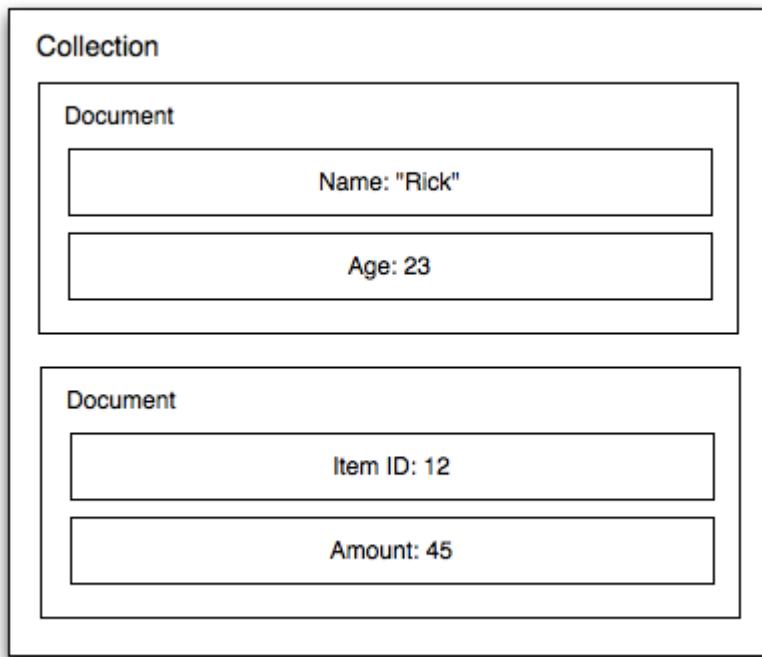


Figure 5.4 If you use MongoDB to store data for your Node application, each item in your MongoDB collection can have a completely different schema

The most mature, actively maintained MongoDB API module is Christian Amor Kvalheim's node-mongodb-native¹³. Install this module using the following npm command:

Footnote 13 <https://github.com/mongodb/node-mongodb-native>

```
npm install mongodb
```

WARNING

Windows installation of node-mongodb-native

Windows installation requires msbuild.exe, which is installed by Microsoft Visual Studio.

CONNECTING TO MONGODB

After installing node-mongodb-native and running your MongoDB server, use the following code to establish a database server connection:

```
var mongodb = require('mongodb')
, server = new mongodb.Server('127.0.0.1', 27017, {});

var client = new mongodb.Db('mtest', server);
```

ACCESSING A MONGODB COLLECTION

Listing 5.15 shows how you can access a collection once the database connection is open. If at any time after completing your database operations you want to close your MongoDB connection, execute `client.close()`.

Listing 5.15 mongo_connect.js: Connecting to a MongoDB collection

```
client.open(function(err) {
  if (err) throw err;
  client.collection('test_insert', function(err, collection) {
    if (err) throw err;
    console.log('We are now able to perform queries.');
  });
});
```

① Put MongoDB query code here

INSERTING A DOCUMENT INTO A COLLECTION

The following code inserts a document in a collection and prints its unique document ID:

```
collection.insert(
{
  "title": "I like cake",
  "body": "It is quite good."
},
{safe: true},
function(err, documents) {
  if (err) throw err;
  console.log('Document ID is: ' + documents[0]._id);
}
);
```

Although you can display `documents[0]._id` using `console.log` and it will display as a string, it's not actually a string. Document identifiers from MongoDB are encoded as in "binary JSON" (BSON). BSON is a data interchange format primarily used by MongoDB. It's used instead of JSON to move data to and from the MongoDB server because it's, in most cases, more space efficient than JSON and can be parsed faster. Taking less space and being easier to scan means database interactions end up faster.

UPDATING DATA USING DOCUMENT IDS

BSON document identifiers may be used to update data. Listing 5.16 shows how to update a document using its ID:

Listing 5.16 mongo_update.js: Updating a MongoDB document

```
var _id = new client.bson_serializer
          .ObjectId('4e650d344ac74b5a01000001');

collection.update(
  {_id: _id},
  {$set: {"title": "I ate too much cake"}},
  {safe: true},
  function(err) {
    if (err) throw err;
  }
);
```

NOTE

Safe mode

You'll notice in each of the previous examples that the option `{safe: true}` is specified. This indicates that you want the database operation to complete before executing the callback. If your callback logic is in any way dependent on the database operation being complete, you'll want to use this option. If your callback logic isn't dependent, then you can get away with using `{ }` instead.

SEARCHING FOR DOCUMENTS

To search for documents in MongoDB, use the `find` method. The following example shows logic that'll display all items in a collection with a title of "I like cake":

```

collection.find({ "title": "I like cake" }).toArray(
  function(err, results) {
    if (err) throw err;
    console.log(results);
  }
);

```

DELETING DOCUMENTS

Want to delete something? You can delete a record by its internal ID (or any other criteria) using code similar to the following:

```

var _id = new client
  .bson_serializer
  .ObjectId('4e6513f0730d319501000001');
collection.remove({_id: _id}, {safe: true}, function(err) {
  if (err) throw err;
});

```

MongoDB is a powerful database and node-mongodb-native offers high-performance access to it, but you may want to use an API that abstracts database access, handling the details for you in the background. This allows you to develop faster, while maintaining less lines of code. The most popular of these APIs is called Mongoose.

5.3.3 Mongoose

LearnBoost’s Mongoose is a Node module that makes using MongoDB painless. Mongoose “models” (in model-view-controller parlance) provide an interface to MongoDB collections as well as additional useful functionality such as schema hierarchy, middleware, and validation. Schema hierarchy allows the association of one model with another, enabling, for example, a blog post to contain associated comments. Middleware allows the transformation of data or the triggering of logic during model data operations, making possible tasks like the automatic pruning of child data when a parent is removed. Mongoose’s validation support lets you determine what data is acceptable at the schema level, rather than having to manually deal with it.

Although we’ll focus solely on the basic use of Mongoose as a datastore, if you decide to use Mongoose in your application you’ll definitely benefit from reading its online documentation¹⁴ and learning all it has to offer.

Footnote 14 <http://mongoosejs.com/>

In this section, we’ll walk you through the basics of Mongoose, including how to:

- Open and close a MongoDB connection
- Register a schema
- Add a task
- Search for a document
- Update a document
- Remove a document

First, let’s install Mongoose via npm using the following command:

```
npm install mongoose
```

OPENING AND CLOSING A CONNECTION

Once you’ve installed Mongoose and have started your MongoDB server, the following example code will establish a MongoDB connection, in this case to a database called “tasks:”

```
var mongoose = require('mongoose')
, db = mongoose.connect('mongodb://localhost/tasks');
```

If at any time in your application you want to terminate your Mongoose-created connection, the following code will close it:

```
mongoose.disconnect();
```

REGISTERING A SCHEMA

When managing data using Mongoose, you'll need to register a schema. The following code shows the registration of a schema for tasks:

```
var Schema = mongoose.Schema;
var Tasks = new Schema({
  project: String,
  description: String
});
mongoose.model('Task', Tasks);
```

Mongoose schemas are powerful. In addition to defining data structures, they also allow you to set defaults, process input, and enforce validation. For more on Mongoose schema definition, see Mongoose's online documentation¹⁵.

Footnote 15 <http://mongoosejs.com/docs/schematypes.html>

ADDING A TASK

Once a schema is registered, you can access it and put Mongoose to work. The following code shows how to add a task using the appropriate model:

```
var Task = mongoose.model('Task');
```

```

var task = new Task();
task.project = 'Bikeshed';
task.description = 'Paint the bikeshed red.';
task.save(function(err) {
  if (err) throw err;
  console.log('Task saved.');
});

```

SEARCHING FOR A DOCUMENT

Searching with Mongoose is similarly easy. The task model's `find` method allows you to find all, or select documents using a JavaScript object to specify your filtering criteria. The following example code searches for tasks associated with a specific project and outputs each task's unique ID and description:

```

var Task = mongoose.model('Task');
Task.find({ 'project': 'Bikeshed' }).each(function(err, task) {
  if (task != null) {
    console.log('ID:' + task._id);
    console.log(task.description);
  }
});

```

UPDATING A DOCUMENT

Although it's possible to use a model's `find` method to zero in on a document that you can subsequently change and save, Mongoose models also have an `update` method expressly for this purpose. Listing 5.17 shows how you can update a document using Mongoose:

Listing 5.17 Updating a document using Mongoose

```

var Task = mongoose.model('Task');
Task.update(
  {_id: '4e65b793d0cf5ca508000001'},
  {description: 'Paint the bikeshed green.'},
  {multi: false},
  function(err, rows_updated) {
    if (err) throw err;
}

```

- ➊ Update using internal ID
- ➋ Only update one document

```

        console.log('Updated.');
    }
);

```

REMOVING A DOCUMENT

It's easy to remove a document in Mongoose once you've retrieved it. You can retrieve and remove a document using its internal ID (or any other criteria if you use the `find` method instead of `findById`) using code similar to the following:

```

var Task = mongoose.model('Task');
Task.findById('4e65b3dce1592f7d08000001', function(err, task) {
  task.remove();
});

```

You'll find much to explore in Mongoose. It's a great all-around tool that enables you to pair the flexibility and performance of MongoDB with the ease of use traditionally associated with relational database management systems.

5.4 Summary

Now that you've gained a healthy understanding of data storage technologies, you have the basic knowledge you need to deal with common application data storage scenarios.

If you're creating multi-user web applications, you'll most likely use a DBMS of some sort. If you prefer the SQL-based way of doing things, MySQL and PostgreSQL are well-supported RDBMS's. If you find SQL limiting in terms of performance or flexibility, Redis and MongoDB are rock-solid options. MongoDB is a great general-purpose DBMS whereas Redis excels in dealing with frequently changing, less complex data.

If you don't need the bells and whistles of a full-blown DBMS and want to avoid the hassle of setting one up, you've several options. If speed and performance are key, and you don't care about data persisting beyond application restarts, in-memory storage may be a good fit. If you aren't concerned about performance and don't need to do complex queries on your data—as with a typical command-line application—storing data to files may suit your needs.

Don't be afraid to use more than one type of storage mechanism in an application. If you were building a content management system, for example, you might store web application configuration options using files, stories using MongoDB, and user-contributed story ranking data using Redis. How you handle persistence is limited only by your imagination.

With the basics of web application development and data persistence under your belt, you've learned the fundamentals you need to make simple web applications. You're now ready to move on to testing, an important tool you'll need to ensure that what you code today works tomorrow.

Testing Node applications



In this chapter:

- Testing logic with Node's Assert module
- Using Node unit testing frameworks
- Simulating and controlling web browsers using Node

As you add features to your application you run the risk of introducing bugs. An application isn't complete without being tested and as manual testing is tedious, and prone to human error, automated testing has become increasingly popular with developers. Automated testing is the idea of writing logic to test your code, rather than running through application functionality by hand.

If the idea of automated testing is new to you, think of it as a robot doing all of the boring stuff for you, allowing you to focus on the interesting stuff. Every time you make a change you can get the robot to make sure bugs haven't crept in. Although you may not have completed or started your first Node application, it's good to get a handle on how to implement automated testing because you will be able to write tests as you develop.

In this chapter we'll look at two types of automated testing: unit testing and acceptance testing. Unit testing tests code logic directly, typically at a function/method level, and is applicable to all types of applications. Unit testing methodology can be divided into two major forms: test-driven development¹ (TDD) and behavior-driven development² (BDD). Practically speaking, TDD and

BDD are largely the same thing where differences mostly lie in the language used to describe the tests as you will see when we go through examples. There are other differences between TDD and BDD too but they are beyond the scope of this book.

Footnote 1 http://en.wikipedia.org/wiki/Test-driven_development

Footnote 2 http://en.wikipedia.org/wiki/Behavior-driven_development

Acceptance testing is an additional layer of testing most commonly used for web applications. Acceptance testing involves scripting control of a browser and attempting to trigger web application functionality with it. We'll look at established solutions for both unit and acceptance testing. For unit testing, we'll cover Node's Assert module and the Mocha, Nodeunit, Vows, and Should.js frameworks. For acceptance testing, we'll look at the Tobi and Soda frameworks. Figure 6.1 places the tools alongside their respective testing methodologies and flavors.

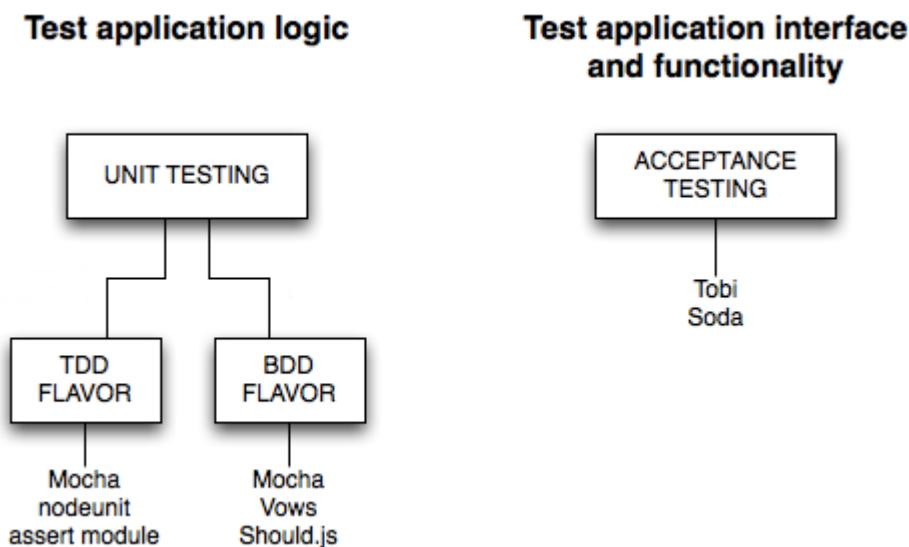


Figure 6.1 Test framework overview

Let's start with unit testing.

6.1 Unit testing

Unit testing is a type of automated testing where you write logic to test discrete parts of your application. Writing tests makes you think more critically about your application design choices and helps avoid pitfalls early. They also give you confidence that your recent changes haven't introduced errors. Although unit tests take a bit of work up front to write, they can save you time by lessening the need to manually retest every time you make a change to an application.

Unit testing can be tricky and asynchronous logic can add new challenges. Asynchronous unit tests can run in parallel so you've got to be careful that tests do not interfere with each other. For example, if your tests create temporary files on disk you'll have to be careful that when you delete the files after a test, you don't delete the working files of another test that hasn't yet finished. For this reason many unit testing frameworks include flow control to sequence the running of tests.

In this section, we'll show you how to use:

- Node's built-in `assert` module: a good building block for TDD-style automated testing
- Nodeunit: a longtime favourite TDD-style testing framework of the Node community
- Mocha: a relatively new testing framework that can be used for TDD or BDD style testing
- Vows: a widely used BBD-style testing framework
- Should.js: a module that builds on Node's `assert` module to provide BDD-style assertions

Let's start with the `Assert` module, which is included with Node.

6.1.1 The Assert module

The basis for most Node unit testing is the built-in `assert` module, which tests a condition and, if the condition isn't met, throws an error. Node's `assert` module is taken advantage of by many third-party testing frameworks, but even without a testing framework you can accomplish a lot of testing with it.

A SIMPLE EXAMPLE

Suppose you have a simple todo application that stores items in memory and you want to assert that it is doing what you think it is doing.

Listing 6.1 defines a module containing the core application functionality. Module logic supports creating, retrieving, and deleting todo items. It also includes a simple `doAsync` method so we can look at testing asynchronous methods too. Let's call this file `todo.js`.

Listing 6.1 todo.js: model for todo list

```
function Todo () {
  this.todos = [];
}
Todo.prototype.add = function (item) {
  if (!item) throw new Error('Todo#add requires an item')
  this.todos.push(item);
}
```

1 Define todo database

2 Add a todo item

3 Delete all todo

```

Todo.prototype.deleteAll = function () {
  this.todos = [];
}
Todo.prototype.getCount = function () {
  return this.todos.length;
}
Todo.prototype.doAsync = function (cb) {
  setTimeout(cb, 2000, true);
}
module.exports = Todo;

```

① Add some todo items
④ Get count of todo items
⑤ Callback with 'true' after 2 secs
⑥ Export Todo function

Now we can use Node's Assert module to test the code.

In a file called test.js enter the following code to load the necessary modules, setup a new todo list, and set a variable that will track testing progress.

Listing 6.2 test.js: Set up necessary modules

```

var assert = require('assert');
var Todo = require('./todo');
var todo = new Todo();
var testsCompleted = 0;

```

USING “EQUAL” TO TEST THE CONTENTS OF A VARIABLE

Next add a test of the todo application's delete functionality.

Note the use of equal in listing 6.3. equal is the assert module's most used assertion and tests the contents of a variable are indeed equal to a value specified in the second argument. In the example code, a todo item is created and then all items are deleted.

Listing 6.3 test.js: Test to make sure that no todo items remain after deletion

```

function deleteTest () {
  todo.add('Delete Me'); ①
  assert.equal(todo.getCount(), 1, '1 item should exist'); ②
  todo.deleteAll(); ③
  assert.equal(todo.getCount(), 0, 'No items should exist'); ④
  testsCompleted++; ⑤
}

```

- ① Add some data in order to test delete

- ② Assert data was added correctly
- ③ Delete all the records
- ④ Assert record was deleted
- ⑤ Notes that test has completed

As there should be no todos, the value of `todo.getCount()` should be 0 if the application logic is working properly. If a problem occurs, an exception is thrown. If the variable `todo.getCount()` isn't set to 0 the assertion would result in a stack trace showing an error message, "No items should exist." After the assertion, `testsCompleted` is incremented to note that a test has completed.

USING "NOTEQUAL" TO FIND PROBLEMS IN LOGIC

Next, add the code in listing 6.4, which is a test of the todo application's add functionality:

Listing 6.4 test.js: Test to make sure adding a todo works

```
function addTest () {
  todo.deleteAll();      ①
  todo.add('Added');    ②
  assert.notEqual(todo.getCount(), 0, '1 item should exist');  ③
  testsCompleted++;    ④
}
```

- ① Delete any existing items
- ② Add item
- ③ Assert that items exist
- ④ Note that test has completed

The `assert` module also allows `notEqual` assertions. This type of assertion is useful when generation of a certain value by application code indicates a problem in logic. Listing 6.4 shows the use of a `notEqual` assertion. All todo items are deleted, an item is added, and the application logic then gets all items. If the number of items is 0, the assertion will fail and an exception will be thrown.

USING ADDITION FUNCTIONALITY: STRICTEQUAL, NOTSTRICTEQUAL, DEEPEQUAL, NOTDEEPEQUAL

In addition to `equal` and `notEqual` functionality, the `assert` module offers strict versions of assertions called `strictEqual` and `notStrictEqual`. These use the strict equality operator ("`==`") rather than the more permissive ("`=`"). To compare objects, the `assert` module offers `deepEqual` and `notDeepEqual`. The "deep" in the names of these assertions indicates that they recursively compare two objects, comparing two object's properties and, if the properties are themselves objects, comparing these as well.

USING "OK" TO TEST FOR A ASYNCRONOUS VALUE BEING TRUE

Next, add a test of the todo application's `doAsync` method as shown in listing 6.5. Since this is an asynchronous test, we are providing a callback function (`cb`) to know when the test has completed since we can't rely on the function returning to tell us like we can with synchronous tests. To see if a callback function is passed the value `true`, we use the `ok` assertion. The `ok` assertion provides an easy way to test a value for being `true`.

Listing 6.5 test.js: Test to see if the doAsync callback is passed true

```
function doAsyncTest (cb) {
  todo.doAsync(function (value) {           ①
    assert.ok(value,'Callback should be passed true');
    testsCompleted++;                     ②
    cb();                                ③
  })
}
```

- ① 2 secs later callback will fire
- ② Assert value is true
- ③ Note that test has completed
- ④ Trigger callback when done

TESTING THAT THROWN ERRORS ARE CORRECT

You can also use the `assert` module to check that thrown error messages are correct, as the following listing 6.6 shows. The second argument in the `throws` call is a regular expression that looks for the text "requires" in the error message.

Listing 6.6 test.js: Test to see if add throws when missing param

```
function throwsTest (cb) {
  assert.throws(todo.add, /requires/); ①
  testsCompleted++; ②
}
```

- ① todo.add called with no arguments
- ② Note that test has completed

ADDING LOGIC TO RUN YOUR TESTS

Now that you've defined the tests, you can add logic to the file to run each of the tests. The logic in listing 6.7 will run each test, print how many tests were run and completed when finished:

Listing 6.7 test.js: Running the tests and reporting test completion

```
deleteTest();
addTest();
throwsTest();
doAsyncTest(function () {
  console.log('Completed ' + testsCompleted + ' tests'); ①
})
```

- ① Indicate completion

You can run the tests with the following command:

```
$ node test.js
```

If the tests don't fail, the script informs you of the number of tests completed. It also can be smart to keep track of tests when they start execution as well as complete to protect against flaws in individual tests. For example, where the test may execute without reaching the assertion.

By using Node's built-in functionality each test case had to include a lot of boilerplate to setup the test (i.e. delete all items) and keep track of progress (i.e. the "completed" counter). All this boilerplate shifts focus away from the primary concern of writing test cases, which is better left to a dedicated framework that

would do the heavy lifting and let one focus on testing business logic. Let's look at how you can make things easier using Nodeunit, a third-party unit testing framework.

6.1.2 Nodeunit

A number of excellent testing frameworks have been created by the Node community. Using a unit testing framework simplifies unit testing. Frameworks generally keep track of how many tests have run and make it easy to run multiple test scripts easily.

We'll start with look at Nodeunit³ as it's a time-tested favorite of Node developers who prefer TDD-flavored testing. Nodeunit provides a command-line tool that will run all of your application's tests and let you know how many pass and fail, saving you from having to implement your own application-specific testing tool. In this section you'll learn how to write tests with Nodeunit that can test both Node application code and client-side code run using a web browser. You'll also learn how Nodeunit deals with the challenge of keeping track of running tests asynchronously.

Footnote 3 <https://github.com/caolan/nodeunit>

INSTALLING NODEUNIT

Enter the following to install Nodeunit:

```
$ npm install -g nodeunit
```

Once complete you'll have a new command available named `nodeunit`. This command is given one or more directories or files, containing tests, as an argument and will run all scripts with the extension ".js" within the directories passed.

TESTING NODE APPLICATIONS WITH NODEUNIT

To add Nodeunit tests to your project, create a directory for them (the directory is usually named "test"). Each test script should populate the `exports` object with tests.

To follow is an example Nodeunit server-side test file:

```
exports.testPony = function(test) {
  var isPony = true;
  test.ok(isPony, 'This is not a pony.');
  test.done();
```

```
}
```

Note that the previous test script doesn't require any modules. Nodeunit automatically includes Node's Assert module's methods in an object that it passes to each function exported by a test script. In our example this object is called `test`.

Once each function exported by the test script has completed, the `done` method should be called. If it isn't called, the test will report a failure of "Undone tests." By requiring this method be called, Nodeunit checks that all tests that were started, were also finished.

It also can be helpful to check that all the assertions fire within a test. Why wouldn't assertions fire? When writing unit tests, the danger always exists that the test logic itself is buggy, leading to false-positives. Logic in the test may be written in such a way that certain assertions don't evaluate. The following example shows how `test.done()` can fire and report success even though one of the assertions hasn't executed:

```
exports.testPony = function(test) {
  if (false) {
    test.ok(false, 'This should not have passed.');
  }
  test.ok(true, 'This should have passed.');
  test.done();
}
```

If you want to safeguard against this, you could manually implement an assertion counter, such as the one shown in listing 6.8:

Listing 6.8 nodeunit_count.js: Manually counting assertions

```
exports.testPony = function(test) {
  var count = 0;          ①
  if (false) {
    test.ok(false, 'This should not have passed.');
    count++;             ②
  }
  test.ok(true, 'This should have passed.');
  count++;              ③
  test.equal(count, 2, 'Not all assertions triggered.'); ④
}
```

```
    test.done();
}
```

- ➊ Count assertions
- ➋ Increment assertion count
- ➌ Increment assertion count
- ➍ Test assertion count

This is tedious. Nodeunit offers a nicer way to do this by using `test.expect`. This method allows you to specify the number of assertions each test should include. The result is less lines of unnecessary code.

```
exports.testPony = function(test) {
  test.expect(2);
  if (false) {
    test.ok(false, 'This should not have passed.');
  }
  test.ok(true, 'This should have passed.');
  test.done();
}
```

In addition to testing Node modules, Nodeunit also allows you to test client-side JavaScript, giving you the ability to use one test harness for your web applications. You can learn about that and more advanced techniques by checking out the online documentation⁴.

Footnote 4 <https://github.com/caolan/nodeunit>

Now that you've learned how to use a TDD-flavored unit testing framework, let's look at how you can incorporate a BDD style of unit testing.

6.1.3 Mocha

Mocha is the newest testing framework you'll learn about in this chapter, and it's an easy framework to grasp. Although it defaults to a BDD style, you can also use it in a TDD style as well. Mocha has a wide variety of features, including global variable leak detection, and, like Nodeunit, supports client-side testing.

SIDE BAR**Global variable leak detection**

You should have little need for global variables that are readable application-wide and it's considered a programming best practice to minimize your use of them. But in JavaScript it's easy to inadvertently create global variables by forgetting to include the `var` keyword when declaring a variable. Mocha helps detect accidental global variable "leaks" by throwing an error when you create a global variable during testing.

If you want to disable global leak detection, run `mocha` with the `--ignored-leaks` command-line option. Alternatively, if you want to allow a select number of globals to be used you can specify them using the `--globals` command-line option followed by a comma-delimited list of allowable global variables.

By default, Mocha tests are described using BDD-flavored functions called `describe`, `it`, `before`, `after`, `beforeEach`, and `afterEach` to define tests and set up logic. Mocha's "TDD" interface replaces the use of `describe` with `suite`; `it` with `test`; `before` with `setup`; and `after` with `teardown`. For our example we will stick with the default "BDD" interface.

TESTING NODE APPLICATIONS WITH MOCHA

Let's dive right in and create a small project called "memdb", a small in-memory database, and use Mocha to test it. First create the directories and files for the project:

```
$ mkdir -p memdb/test
$ cd memdb
$ touch index.js
$ touch test/memdb.js
```

The "test" directory is where the tests will live, but before you write any tests you need to install mocha:

```
$ npm install -g mocha
```

By default, Mocha will use the BDD "interface". Listing 6.9 shows you what it looks like:

Listing 6.9 test/memdb.js: Basic structure for Mocha test

```
var memdb = require('..');
describe('memdb', function(){
  describe('.save(doc)', function(){
    it('should save the document', function(){
      });
    });
  });
});
```

Mocha also supports "tdd", "qunit", and "exports" style interfaces which are detailed on the project's site <http://visionmedia.github.com/mocha>, but to illustrate the concept of different interfaces here's the "exports" interface:

```
module.exports = {
  'memdb': {
    '.save(doc)': {
      'should save the document': function(){
        }
      }
    }
}
```

All of these interfaces provide the same functionality, but for now let's stick to the "bdd" interface, and write the first test, listing 6.10, using Node's Assert module to perform the assertions:

Listing 6.10 test/memdb.js: describing the memdb.save functionality

```
var memdb = require('..');
var assert = require('assert');
describe('memdb', function(){①
  describe('.save(doc)', function(){②
    it('should save the document', function(){
      var pet = { name: 'Tobi' };
      memdb.save(pet);
      var ret = memdb.first({ name: 'Tobi' });
③
      assert(ret == pet);④
    })
  })
});
```

- ① describe memdb functionality
- ② describe .save() method's functionality
- ③ describe the expectation
- ④ ensure that the pet was found

To run the tests all you need to do is execute `mocha`. Mocha will look in the `./test` directory by default for JavaScript files to execute. Since you haven't implemented the `.save()` method yet you'll see that the single test defined fails as shown in figure 6.2:

```
wavded@dev ~/Projects/memdb» mocha
.
✖ 1 of 1 test failed:

1) memdb .save(doc) should save the document:
  TypeError: Object #<Object> has no method 'save'
    at Context.<anonymous> (/home/wavded/Projects/memdb/test/memdb.js:8:13)
    at Test.Runnable.run (/usr/local/lib/node_modules/mocha/lib/runnable.js:184:32)
    at Runner.runTest (/usr/local/lib/node_modules/mocha/lib/runner.js:300:10)
    at Runner.runTests.next (/usr/local/lib/node_modules/mocha/lib/runner.js:346:12)
    at next (/usr/local/lib/node_modules/mocha/lib/runner.js:228:14)
    at Runner.hooks (/usr/local/lib/node_modules/mocha/lib/runner.js:237:7)
    at next (/usr/local/lib/node_modules/mocha/lib/runner.js:185:23)
    at Runner.hook (/usr/local/lib/node_modules/mocha/lib/runner.js:205:5)
    at process.startup.processNextTick.process._tickCallback (node.js:244:9)

wavded@dev ~/Projects/memdb» _
```

Figure 6.2 Failing test in Mocha

Let's make it pass! Add listing 6.11 to `index.js`:

Listing 6.11 index.js: added save functionality

```
var db = [];
exports.save = function(doc){
  db.push(doc); ①
};

exports.first = function(obj) {
  return db.filter(function(doc){ ②
    for (var key in obj) {
      if (doc[key] != obj[key]) { ③
        return false;
      }
    }
    return true; ④
  }).shift(); ⑤
}
```

```
};
```

- ① add the doc to our database array
- ② select docs that match every property in obj
- ③ not a match, return false and don't select this doc
- ④ they all matched, return and select the doc
- ⑤ we only want the first doc or null

Run the tests again with mocha and see green, as shown in figure 6.3!:

```
wavded@dev ~/Projects/memdb» mocha
.
✓ 1 test complete (2ms)
```

Figure 6.3 Successful test in Mocha

DEFINING SETUP AND CLEANUP LOGIC USING MOCHA "HOOKS"

This test-case makes the assumption that `memdb.first()` functions appropriately, so you'll want to add a few test-cases for it as well, with expectations defined using the `it()` function. The revised test file, listing 6.12, includes a new concept, the concept of Mocha "hooks". For example the "bdd" interface exposes `beforeEach()`, `afterEach()`, `before()`, and `after()` which take callbacks, allowing you to define setup and cleanup logic before and after test-cases and suites defined with `describe()`.

Listing 6.12 test/memdb.js: Adding beforeEach "hook"

```
var memdb = require('..');
var assert = require('assert');
describe('memdb', function(){
  beforeEach(function(){
    memdb.clear(); ①
  })
  describe('.save(doc)', function(){
    it('should save the document', function(){
      var pet = { name: 'Tobi' };
      memdb.save(pet);
      var ret = memdb.first({ name: 'Tobi' });
      assert(ret == pet);
    })
  })
})
```

```

describe('.first(obj)', function(){
  it('should return the first matching doc', function(){
    var tobi = { name: 'Tobi' };
    var loki = { name: 'Loki' };
    memdb.save(tobi); ③
    memdb.save(loki);
    var ret = memdb.first({ name: 'Tobi' }); ④
    assert(ret == tobi);
    var ret = memdb.first({ name: 'Loki' });
    assert(ret == loki);
  })
  it('should return null when no doc matches', function(){
    var ret = memdb.first({ name: 'Manny' });
    assert(ret == null);
  })
})
})

```

- ① Clear database before each test-case to keep the tests stateless
- ② the first expectation for .first()
- ③ save two documents
- ④ make sure each one can be returned properly
- ⑤ the second expectation for .first()

Ideally test-cases share no state whatsoever. To achieve this with memdb you simply need to remove all the documents, by implementing the `.clear()` method:

```

exports.clear = function(){
  db = [];
};

```

Running mocha again should show you that three tests have passed.

TESTING ASYNCHRONOUS LOGIC

One thing we haven't yet dealt with in our Mocha testing is test asynchronous logic. To show how this is done, let's make a small change to one of the functions we defined earlier in `index.js`. By changing the `save` function to the following, a callback can be optionally provided that will execute after a small delay (meant to simulate some sort of asynchronous operation).

```

exports.save = function(doc, cb){

```

```

db.push(doc);
if (cb) {
  setTimeout(function() {
    cb();
  }, 1000);
}
;

```

Mocha test-cases can be defined as `async` simply by adding an argument to a function defining testing logic. The argument is commonly named "done". Listing 6.13 shows how you could change the initial `.save()` test to work with asynchronous code:

Listing 6.13 memdb_async.js: Testing asynchronous logic

```

describe('.save(doc)', function(){
  it('should save the document', function(done){
    var pet = { name: 'Tobi' };
    memdb.save(pet, function(){ ❶
      memdb.first({ name: 'Tobi' }, function(err, ret){ ❷
        assert(ret == pet); ❸
        done(); ❹
      });
    });
  });
});

```

- ❶ save the doc
- ❷ callback is invoked with the first doc
- ❸ assert that the doc fetches matches "pet"
- ❹ tell mocha we're done with this test-case

This same rule applies to all of the "hooks", for example the `beforeEach()` hook to clear the database would add a callback, and Mocha will wait until it's called in order to move on. If `done()` is invoked with an error as the first argument then Mocha will report the error and mark the hook or test-case as a failure.

```

beforeEach(function(done){
  memdb.clear(done);
})

```

For more about Mocha, check out its full online documentation⁵. Mocha also works for client-side JavaScript like NodeUnit.

Footnote 5 <http://visionmedia.github.com/mocha>

SIDE BAR **Mocha's use of non-parallel testing**

Mocha executes tests one after another rather than in parallel, which makes running test suites execute more slowly, but makes writing tests easier. However, Mocha won't let any test run inordinately long. Mocha, by default, will only allow any given test to run for 2,000 milliseconds before failing it. If you have tests that take longer you can run Mocha with the `--timeout` command-line option and then specify a larger number (or 0 if you want to disable the timeout feature completely). For most testing running tests serially is fine. If you find this problematic, there are other frameworks that execute in parallel like Vows which is also covered in this chapter.

6.1.4 Vows

The tests you can write using the Vows unit testing framework are more structured than many frameworks, with the structure intended to make the tests easy to read and maintain.

Vows uses its own BDD-flavored terminology to define test structure. In the realm of Vows, a test suite contains one or more "batches." A batch can be thought of as a group of related "contexts," or conceptual areas of concern that you want to test. The "batches" and "contexts" run in parallel. A "context" may contain a number of things: a "topic," one or more "vows," and/or one or more related "contexts"⁶. A topic is testing logic related to a context. A vow is a test of the result of a topic. Figure 6.4 visually represents how Vows structures tests.

Footnote 6 Inner "contexts" also run in parallel

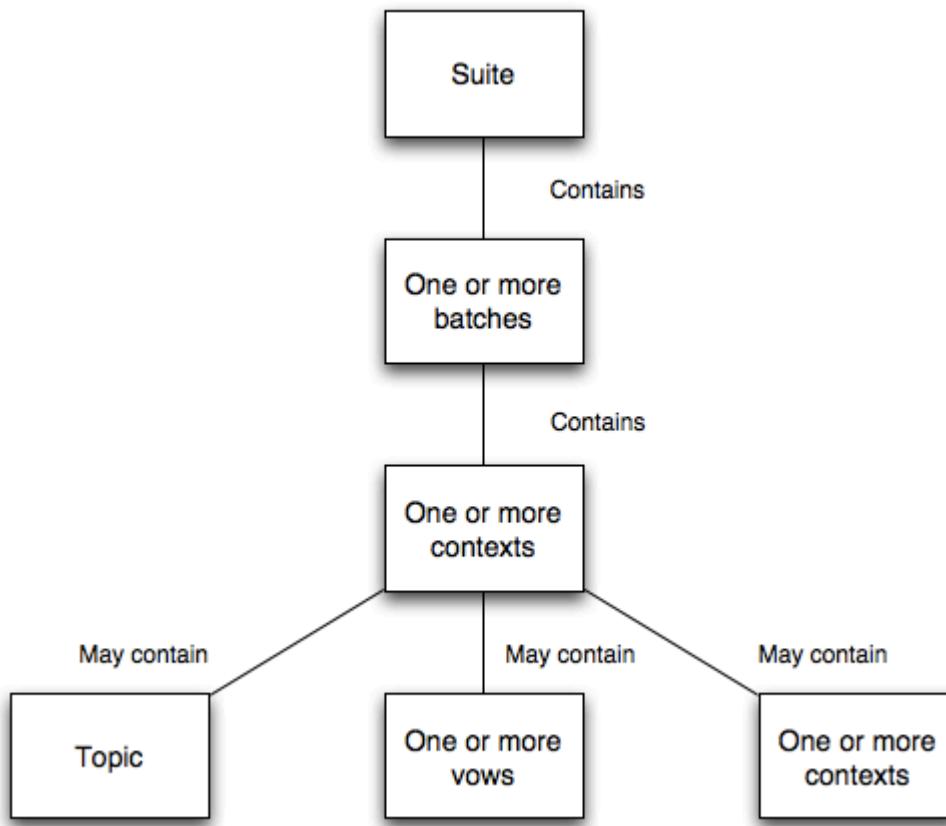


Figure 6.4 Vows can structure tests in a suite using batches, contexts, topics, and vows

Vows, as with Nodeunit and Mocha, is geared towards automated application testing. The difference is primarily in flavor and parallelism, with Vows tests requiring a specific structure and terminology. In this section we'll run through an example application test and explain how you can use the Vows test to run multiple tests at the same time.

Typically, you would install Vows globally to give you access to the `vows` command-line test running tool. Enter the following install Vows:

```
$ npm install -g vows
```

TESTING APPLICATION LOGIC WITH VOWS

You can trigger testing in Vows either by running a script containing test logic or by using the `vows` command-line test runner. The following example of a stand-alone test script (which can be run like any other Node script) uses one of the tests of the todo application's core logic.

Listing 6.14 creates a "batch". Within the "batch," you define a "context." Within the "context," you define a "topic" and a "vow." Note how the code makes use of the callback to deal with asynchronous logic in the "topic." If a "topic" isn't asynchronous, a value can be returned rather than sent via a callback.

Listing 6.14 vows_test.js: Using Vows to test the todo application

```
var vows = require('vows')
var assert = require('assert')
var Todo = require('./todo');

vows.describe('Todo').addBatch({
  'when adding an item': {
    'topic': function () {
      var todo = new Todo();
      todo.add('Feed my cat');
      return todo;
    },
    'it should exist in my todos': function(er, todo) {
      assert.equal(todo.getCount(), 1);
    }
  }
}).run();
```

- 1 A "batch"
- 2 A "context"
- 3 A "topic"
- 4 A "vow"

If you want to include the previous code listing 6.14 in a folder of tests where it could be run with the Vows test runner, you'd change the last line to the following:

```
...  
}).export(module);
```

To run all tests in a folder named "test," enter the following:

```
$ vows test/*
```

For more about Vows, check out the project's online documentation⁷, as shown in figure 6.5:

Footnote 7 <http://vowsjs.org/>

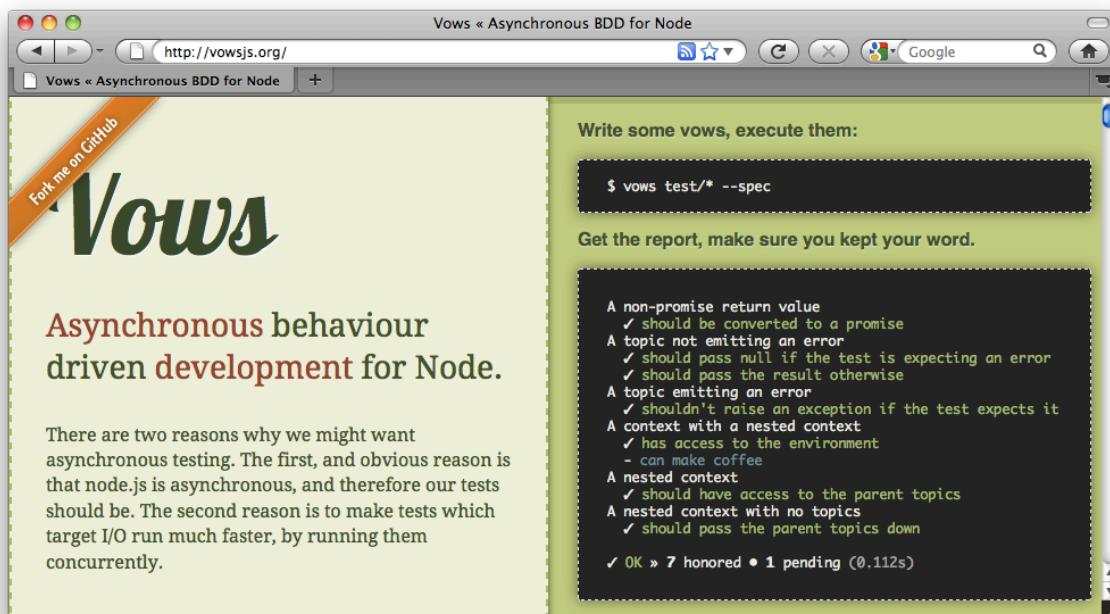


Figure 6.5 Vows combines full-featured BDD-testing capabilities with features such as macros and flow control

Vows offers a comprehensive testing solution, but you might not like the test structure it imposes, as Vows requires the use of “batches,” “contexts,” “topics,” and “vows.” Or you might like the features of a competing testing framework or be familiar with another framework and see no need to learn Vows. If this sounds like you, Should.js might be a good alternative to explore. Rather than being yet another testing framework, Should.js offers a BDD-flavored alternative to using Node’s Assert module.

6.1.5 **Should.js**

Should.js is an assertion library that can help make your tests easier to read by allowing you to express assertions in more of a BDD style. It’s designed to be used in conjunction with other testing frameworks, which lets you continue to use your own framework. In this section you’ll learn how to write assertions with Should.js and will, as an example, write a test for a custom module.

Should.js is easy to use with other frameworks because it augments the `Object.prototype` with a single property: `should`. This allows you to write expressive assertions such as `user.role.should.equal("admin")`, or `users.should.include("rick")`.

TESTING MODULE FUNCTIONALITY USING SHOULD.JS

Let's say you're writing a Node command-line tip calculator that you want to use to figure out who should pay what amount when you split a bill with friends. You'd like to write tests for your calculation logic in a way that's easily understandable to your non-programmer friends, because then they won't think you're cheating them.

To set up your tip calculator application, enter the following commands to set up a folder for it and install Should.js for testing:

```
$ mkdir -p tips/test
$ cd tips
$ touch index.js
$ touch test/tips.js
```

Now lets install Should.js by running:

```
$ npm install should
```

Next, create an `index.js` file, which will contain the logic defining the application's core functionality. Specifically, the tip calculator logic includes four helper functions:

- `addPercentageToEach`, which increases each number in an array by a given percentage
- `sum`, which calculates the sum of each element in an array
- `percentFormat`, which formats a percentage for display
- `dollarFormat`, which formats a dollar value for display

Add this logic by populating `index.js` with the contents of listing 6.15.

Listing 6.15 index.js: Logic for calculating tips when splitting a bill

```
exports.addPercentageToEach = function(prices, percentage) { ①
  return prices.map(function(total) {
    total = parseFloat(total);
    return total + (total * percentage);
  });
}

exports.sum = function(prices) { ②
  return prices.reduce(function(currentSum, currentValue) {
```

```

        return parseFloat(currentSum) + parseFloat(currentValue);
    })
}

exports.percentFormat = function(percentage) { ③
    return parseFloat(percentage) * 100 + '%';
}
exports.dollarFormat = function(number) { ④
    return '$' + parseFloat(number).toFixed(2);
}

```

- ① Add percentage to array elements
- ② Calculate sum of array elements
- ③ Format percentage for display
- ④ Format dollar value for display

Now create the test script in `test/tips.js` in listing 6.16 for it. The script loads our tip logic module, defines a tax and tip percentage and the bill items to test, tests the addition of a percentage to each array element, and tests the bill total.

Listing 6.16 test/tips.js: Logic that calculates tips when splitting a bill

```

var tips = require('..'); ①
var should = require('should');

var tax = 0.12; ②
var tip = 0.15;

var prices = [10, 20]; ③
var pricesWithTipAndTax = tips.addPercentageToEach(prices, tip + tax);
pricesWithTipAndTax[0].should.equal(12.7); ④
pricesWithTipAndTax[1].should.equal(25.4);
var totalAmount = tips.sum(pricesWithTipAndTax).toFixed(2);
totalAmount.should.equal('38.10'); ⑤
var totalAmountAsCurrency = tips.dollarFormat(totalAmount);
totalAmountAsCurrency.should.equal('$38.10');
var tipAsPercent = tips.percentFormat(tip);
tipAsPercent.should.equal('15%');

```

- ① Use tip logic module
- ② Define tax and tip rates
- ③ Define bill items to test
- ④ Test tax and tip addition
- ⑤ Test bill totalling

Run the script using the following command. If all is well, the script should generate no output because no assertions have been thrown and your friends will

be reassured of your honesty.

```
$ node test/tips.js
```

Should.js supports many types of assertions--everything from assertions that use regular expressions to assertions that check object properties--allowing comprehensive testing of data and objects generated by your application. The project's GitHub page⁸ provides comprehensive documentation of Should.js' functionality.

Footnote 8 <http://github.com/visionmedia/should.js>

Having looked at tools designed for unit testing, let's move on to a different style of testing altogether: acceptance testing.

6.2 Acceptance testing

Acceptance testing, also called functional testing, tests outcome, not logic. After you've created a suite of unit tests for your project, acceptance testing would provide an additional level of protection against bugs that unit testing might not detect.

Acceptance testing is similar, conceptually, to testing by end users who follow a list of things to test. But being automated makes acceptance testing fast and doesn't require human labor.

Acceptance testing also deals with complications created by client-side JavaScript behavior. If there's a serious problem created by client-side JavaScript, server-side unit testing won't catch it but thorough acceptance testing will. For example, your application may make use of client-side JavaScript form validation. Acceptance testing will ensure that your validation logic works, rejecting and accepting input appropriately. Or, for another example, you may have Ajax-driven administration functionality—such as the ability to browse content to select featured content for a website's front page—that should only be available to authenticated users. To deal with this, you could write a test to insure the Ajax request produces expected results when the user logs in, and write another test to make sure that those who aren't authenticated can't access this data.

In this section you'll learn how to use two acceptance testing frameworks: Tobi and Soda. Soda provides the benefit of harnessing real browsers for acceptance testing, whereas Tobi, which we'll look at next, is easier to learn and to get up and

running on.

6.2.1 *Tobi*

*Tobi*⁹ is an easy-to-use acceptance testing framework that emulates the browser and takes advantage of `should.js`, offering access to its assertion capabilities. The framework uses two third-party modules, `jsdom` and `htmlparser`, to simulate a web browser, allowing access to a virtual DOM.

Footnote 9 <https://github.com/LearnBoost/tobi>

Tobi enables you to painlessly write tests that will log into your web application, if need be, and send web requests that emulate someone using your application. If *Tobi* returns unexpected results your test can then alert you to the problem.

Because *Tobi* must emulate a user's activities and check the results of web requests, it must often manipulate or examine DOM elements. In the world of client-side JavaScript development, web developers often use the `jQuery`¹⁰ library when they need to interact with the DOM. Developers can also use `jQuery` on the server-side, and *Tobi*'s use of `jQuery` minimizes the amount of learning required to create tests with it.

Footnote 10 <http://jquery.com>

In this section we'll talk about how you can use *Tobi* to test any running web application, including non-Node applications, over the network. We'll also show you how to use *Tobi* to test web applications created with Express, even if the Express-based web application isn't running.

TESTING WEB APPLICATIONS WITH TOBI

If you'd like to create tests using *Tobi*, first create a directory for them (or use an existing application directory), then change to the directory in the command-line, and enter the following to install *Tobi*:

```
$ npm install tobi
```

Listing 6.17 is an example of using *Tobi* to test the login functionality of a website—in this case running the todo application we tested earlier in the chapter. The test attempts to create a todo item, then looks for it on the response page. If you run the script using Node and no exceptions are thrown, the test passed.

The script creates a simulated browser, uses it to perform an HTTP GET

request for a login form, fills in the form’s name and password fields, then submits the form. The script then checks the contents of a `div` with the class “messages.” If the `div` contains the text “Login successful” then the test passes.

Listing 6.17 tobi_remote_test.js: Testing a remote site’s login capability using Tobi

```
var tobi = require('tobi');
var browser = tobi.createBrowser(3000, '127.0.0.1'); 1
browser.get('/', function(res, $){ 2
  $('form')
    .fill({ item: 'Floss the cat' }) 3
    .submit(function(res, $) { 4
      $('ul')
        .html()
        .indexOf('Floss the cat')
        .should.not.equal(-1);
    });
});
```

- 1** Create browser
- 2** Get todo form
- 3** Fill in form
- 4** Submit data

If you want to test an Express application you’ve built, rather than a remote site, it’s similarly straightforward. Listing 6.18 is an example one-page Express site. Note that the last line populates the `exports` object with the application object. This allows Tobi to use `require` to access the application for testing.

Listing 6.18 app.js: Example Express web application

```
var express = require('express');
var app = express.createServer();
app.get('/about', function(req, res) {
  res.send( 1
    + '<html>'
    + '  <body>'
    + '    <div>'
    + '      <h1>About</h1>'
    + '    </div>'
    + '  </body>'
    + '</html>'
  );
});
```

```
});  
module.exports = app; ②
```

- ① Send HTML response
- ② Expose "app" when file is imported as a module

You can test the previous application without even running it. The following Tobi test shows how you'd do this:

```
var tobi = require('tobi');
var app = require('./app');
var browser = tobi.createBrowser(app);
browser.get('/about', function(res, $){
  res.should.have.status(200);
  $('div').should.have.one('h1', 'About');
  app.close();
});
```

Tobi doesn't include a test runner, but you can use it with unit testing frameworks such as Mocha or Nodeunit.

6.2.2 Soda

Soda¹¹ takes a different approach to acceptance testing. Whereas other Node acceptance testing frameworks simulate browsers, Soda remote controls real browsers. Soda, as shown in figure 6.6, does this by sending instructions to the Selenium Server (also known as Selenium RC), or the Sauce Labs "Sauce Cloud" on-demand testing service.

Footnote 11 <https://github.com/LearnBoost/soda>

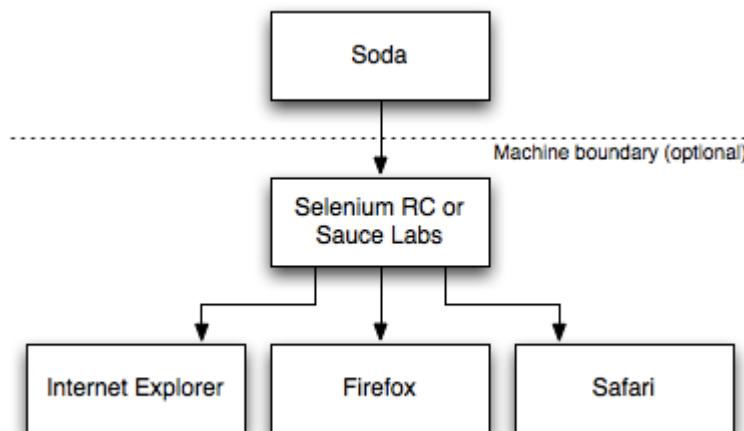


Figure 6.6 Soda is an acceptance testing framework that allows real browsers to be remote controlled. Whether using Selenium RC or the Sauce Labs service, Soda provides an API that allows a Node to perform direct testing that takes into account the realities of different browser implementations.

Selenium Server will open browsers on the machine on which it's installed, whereas Sauce Cloud will open virtual ones on a server somewhere on the internet. Selenium Server and Sauce Cloud, rather than Soda, do the talking to the browsers, but they relay any requested info back to Soda. If you want to do a number of tests in parallel and not tax your own hardware, consider using Sauce Cloud.

In this section you'll learn how to install Soda and the Selenium Server, how to test applications with Soda and Selenium, and how to test applications with Soda and Sauce Cloud.

INSTALLING SODA AND THE SELENIUM SERVER

To do testing with Soda you need to install the soda npm package and the Selenium Server (if you're not using Sauce Labs). Enter the following to install Soda:

```
$ npm install soda
```

Selenium Server requires Java to run. If Java isn't installed, please see consult the official Java download page¹² for instructions specific to your operating system.

Footnote 12 <http://www.java.com/en/download/>

Installing Selenium Server is fairly straightforward. All you have to do is download a recent ".jar" file from the Selenium Downloads page¹³. Once you've downloaded the file you can run the server with the following command (although the filename will likely contain a different version number):

Footnote 13 <http://seleniumhq.org/download/>

```
java -jar selenium-server-standalone-2.6.0.jar
```

TESTING WEB APPLICATIONS WITH SODA AND SELENIUM

Once you have the server running you can include the following code in a script to set up for running tests. In the call to `createClient` the `host` and `port` indicate the host and port used to connect to the Selenium Server. By default these should be "127.0.0.1" and "4444" respectively. The `url` in the call to `createClient` specifies the base URL that you should open in the browser for testing and the `browser` specifies the browser to be used for testing.

```
var soda = require('soda')
var assert = require('assert');
var browser = soda.createClient({
  host: '127.0.0.1',
  port: 4444,
  url: 'http://www.reddit.com',
  browser: 'firefox'
});
```

In order to get feedback on what your testing script is doing, you may want to include the following code. This code prints each Selenium command as it's attempted.

```
browser.on('command', function(cmd, args){
  console.log(cmd, args.join(', '));
});
```

Next in your test script should be the tests themselves. Listing 6.19 is an example test that attempts to log a user into Reddit and fails if the text "logout" isn't present on the resulting page. Commands like "clickAndWait" are documented on the Selenium website¹⁴.

Footnote 14 <http://release.seleniumhq.org/selenium-core/1.0.1/reference.html>

Listing 6.19 soda_test.js: A Soda test allows you to string commands together to control the actions of a browser

```
browser
  .chain ①
  .session() ②
  .open('/') ③
```

```

.type('user', 'mcantelon')    ④
.type('passwd', 'mahsecret')
.clickAndWait('//button[@type="submit"]')  ⑤
.assertTextPresent('logout')  ⑥
.testComplete()   ⑦
.end(function(err){          ⑧
  if (err) throw err;
  console.log('Done!');
});

```

- ① Enable method chaining
- ② Start Selenium session
- ③ Open URL
- ④ Enter text into form field
- ⑤ Click button and wait
- ⑥ Make sure text exists
- ⑦ Mark test as complete
- ⑧ End Selenium session

TESTING WEB APPLICATIONS WITH SODA AND SAUCE CLOUD

If you go the Sauce Cloud route, sign up for the service at the Sauce Labs¹⁵ website and change the code in your test script that returns `browser` to something like what you see in listing 6.20:

Footnote 15 <https://saucelabs.com>

Listing 6.20 soda_cloud.js: Using Soda to control a Sauce Cloud browser

```

var browser = soda.createSauceClient({
  'url': 'http://www.reddit.com/',
  'username': 'yourusername',  ①
  'access-key': 'youraccesskey',  ②
  'os': 'Windows 2003',  ③
  'browser': 'firefox',  ④
  'browser-version': '3.6',  ⑤
  'name': 'This is an example test',
  'max-duration': 300  ⑥
});

```

- ① Sauce Labs user name
- ② Sauce Labs API key
- ③ Desired operating system
- ④

- ④ Desired browser type
- ⑤ Desired browser version
- ⑥ Make test fail if it takes too long

And that's it. You've now learned the fundamentals of a powerful testing method that can complement your unit tests and make your applications much more resistant to accidentally added bugs.

6.3 Summary

By incorporating automated testing into your development, you greatly decrease the odds of bugs creeping into your codebase and you can develop with greater confidence.

If you're new to unit testing, Mocha and Nodeunit are excellent frameworks to start with: they're easy to learn, flexible, and can work with Should.js if you want to run BDD-style assertions. If you like the BDD approach and seek a system for structuring tests and controlling flow, Vows also may be a good choice.

In the realm of acceptance testing, Tobi is a great place to start. Tobi is easy to set up and use, and if you're familiar with jQuery you'll be up and running quickly. If your needs require acceptance testing in order to take into account browser discrepancies, Soda may be worth running, but testing is slower and you must learn Selenium API.

Now that you've a handle on how automated testing can be conducted in Node, you're ready to learn about Connect, an HTTP middleware framework that helps you avoid reinventing the wheel in your web applications.



Connect

In this chapter:

- Setting up a Connect application
- How Connect middleware work
- Why middleware ordering matters
- Mounting middleware & servers
- Creating configurable middleware
- Error handling middleware

Connect is a framework which uses modular components called "middleware" to implement web application logic in a reusable manner. In Connect, middleware is simply a function that intercepts the request and response objects provided by the HTTP server, executes logic, and when its finished either ends the response or passes it to the next middleware. Connect "connects" the middleware together using what's called the "dispatcher."

Connect allows you to write your own middleware but also includes several common components that can be used in your applications such as: request logging, static file serving, request body parsing, and session managing. It serves as an abstraction layer for developers who wish to build their own higher-level web frameworks since Connect can be easily expanded and built upon. In figure 7.1, you can see how a Connect application is composed of the "dispatcher," as well as any arrangement of middleware.

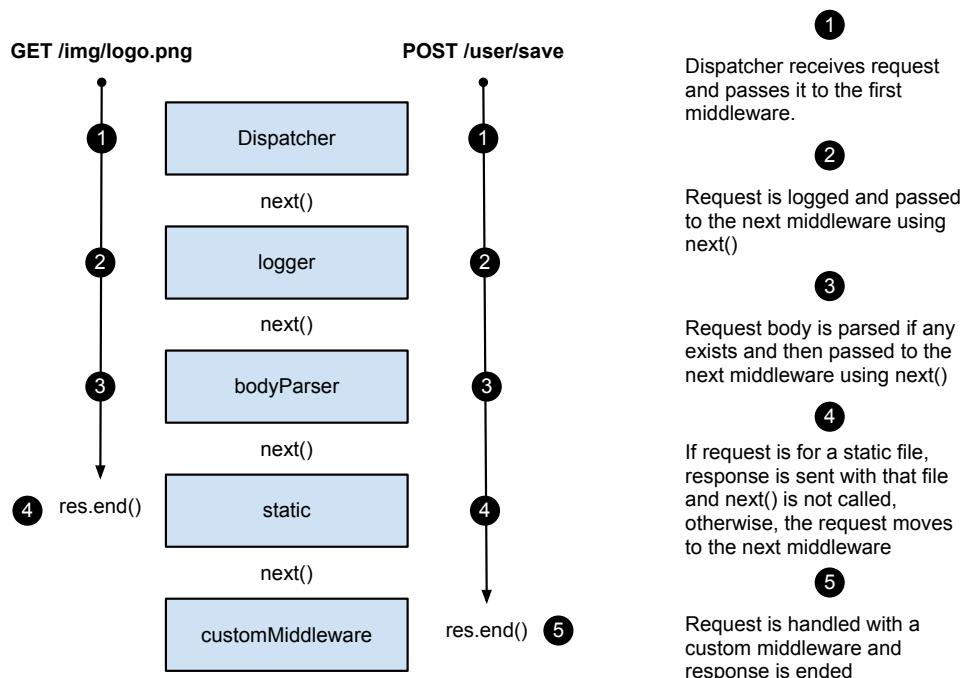


Figure 7.1 The lifecycle of two HTTP requests making their way through a Connect server

SIDE BAR

Connect and Express

The concepts discussed in this chapter are directly applicable to the higher-level framework Express because it extends and builds upon Connect with additional higher-level "sugar". After reading this chapter you'll have a firm understanding of how Connect middleware work and how to compose them together to create an application. In a later chapter, we'll be using Express to make writing web applications more enjoyable with a higher-level API than Connect provides. In fact, much of the functionality that Connect now provides originated in Express, before the abstraction was made (leaving lower-level "building blocks" to Connect and the expressive "sugar" to Express).

To start off, let's create a basic Connect application.

7.1 Setting up a Connect application

Since Connect is a third party module, it is not included by default when you install Node. To install Connect, you can download it from the npm registry using the command shown here:

```
$ npm install connect
```

Now that installing is out of the way you can begin learning Connect from the ground up, beginning with creating a basic Connect application. To do this, start off by requiring the `connect` module which is a function that returns a bare Connect "application" when invoked.

If you remember from Chapter 4, we discussed how `http.createServer()` accepts a callback function to act on incoming requests. The "application" that Connect creates is actually a JavaScript function designed to take the HTTP request and dispatch it to the middleware you have specified.

Listing 7.1 shows what the minimal Connect application looks like. This bare application has no middleware added to it, so any HTTP request that it receives will respond with a "404 Not Found" status code by the dispatcher.

Listing 7.1 `minimal_connect.js`: A minimal Connect application

```
var connect = require('connect');
var app = connect();
app.listen(3000);
```

When you fire up the server and send it an HTTP request (with `curl` or a web browser) you will see the text "Cannot GET /" indicating that this application is not configured to handle the requested url. This is the first example of how Connect's dispatcher works: it invokes each attached middleware one by one until one of them decides to respond to the request. If it gets to the end of the list of middleware and none of them respond, then the application will respond with a 404.

Now that you have learned how to create a barebones Connect app and how the dispatcher works, we will take a look at how to make the application actually *do something* by defining and adding middleware.

7.2 How Connect middleware work

In Connect, middleware is a simply a JavaScript function that by convention accepts three arguments: a request object, a response object, and an argument commonly named `next`, which is a callback function indicating that the middleware is done and the next middleware can be executed.

The concept of middleware was initially inspired by Ruby's "Rack" framework,

providing a very similar modular interface, however due to the streaming nature of Node the API is not identical. Middleware are great because they are designed to be small, self-contained, and reusable across applications.

In this section you will learn the basics of middleware by taking that barebones Connect application from before and building functionality on top of two simple "layers" of middleware that together make up the app:

- A `logger` middleware, to log requests to the console
- A `hello` middleware, to respond to the request with "hello world"

7.2.1 Middleware that does logging

Suppose you wanted to create a log file of the request method and URL of incoming requests from your server. To do this you would create a function, in this case we'll call it `logger`, which accepts the request and response objects, as well as the `next` callback function.

The `next` function may be called from within middleware to tell the dispatcher that the middleware has done its business and control may be passed to the next middleware. A callback function, rather than the method returning, is used so that asynchronous logic can be ran within the middleware and the dispatcher will only move onto the next middleware after it has been completed. Using `next()` a nice mechanism to handle the flow between middleware.

For the `logger` middleware, you could invoke `console.log()` with the request method and url, outputting something like "GET /user/1", then invoke the `next()` function to invoke the subsequent middleware.

```
function logger(req, res, next) {
  console.log('%s %s', req.method, req.url);
  next();
}
```

And there you have it, a perfectly valid middleware that prints out the request method and url of each HTTP request received, and then calls `next()` to pass control back to the dispatcher. To use this middleware in the application, invoke the `.use()` method, passing it the middleware function:

```
var connect = require('connect');
var app = connect();
```

```
app.use(logger);
app.listen(3000);
```

After issuing a few requests to your server (again you can use curl or a web browser) you will see output similar to the following on your console:

```
GET /
GET /favicon.ico
GET /users
GET /user/1
```

But logging requests is just one of the layers of middleware that makes up your application. You still have to send some sort of response to the client. So that will come in your next middleware.

7.2.2 A middleware that responds with "hello world"

The second middleware in this app will actually send a response to the HTTP request. It is the exact same code as the "hello world" example server callback function on the Node homepage:

```
function hello(req, res) {
  res.setHeader('Content-Type', 'text/plain');
  res.end('hello world');
}
```

You can use this second middleware with your `app` by invoking the `.use()` method, which can be called any number of times to add multiple middleware. Listing 7.2 ties the whole app together. The addition of the `hello` middleware in the following listing will make the server first invoke the `logger` which prints text to the console, and then respond to every HTTP request with the text "hello world".

Listing 7.2 multiple_connect.js: Multiple Connect middleware

```

var connect = require('connect');
function logger(req, res, next) {
  console.log('%s %s', req.method, req.url);
  next();
}
function hello(req, res) {
  res.setHeader('Content-Type', 'text/plain');
  res.end('hello world');
}
connect()
  .use(logger)
  .use(hello)
  .listen(3000);

```

1 prints out the HTTP method and request URL and calls next()
2 ends response to the HTTP request with "hello world"

In this case, the `hello` middleware does not have a "next" callback argument. This is because this middleware finishes the HTTP response and therefore never needs to give control back the dispatcher. For cases like this, the "next" callback is optional, which is convenient because it matches the signature of the `http.createServer` callback function. So if you have already written an HTTP server using just the `http` module, then you already have a perfectly valid middleware that you can reuse in your Connect application.

The `use()` function returns an instance of a Connect application to support method chaining, as shown previously. Note that chaining the `.use()` calls is not required, and the following snippet is functionally equivalent:

```

var app = connect();
app.use(logger);
app.use(hello);
app.listen(3000);

```

Now that you have a simple "hello world" application working, we'll cover why the ordering of middleware `.use()` calls is important, and how you should use the ordering strategically to alter how your application works.

7.3 Why middleware ordering matters

Connect tries to be unassuming in order to maximize flexibility for application and framework developers. One example of this is the flexibility to define the order in which middleware are executed, a simple concept, but one often overlooked.

In this section you will see how the ordering of middleware in your application dramatically affects the way it behaves. Specifically, we will cover how:

- Middleware will stop the execution of remaining middleware when `next()` is omitted
- You can use the powerful middleware-ordering feature to your advantage
- You can leverage middleware to perform authentication

7.3.1 When middleware doesn't call `next()`

Consider the previous "hello world" example, where the `logger` middleware is used first, followed by the `hello` middleware second. Connect will invoke the logging middleware before the other; logging to `stdout` and then responding to the HTTP request. But consider what would happen if the ordering were to be switched as shown in listing 7.3.

Listing 7.3 `hello_before_logger.js`: Incorrectly placing the 'hello' middleware before the 'logger' middleware

```
var connect = require('connect');
function logger(req, res, next) { ①
  console.log('%s %s', req.method, req.url);
  next();
}
function hello(req, res) { ②
  res.setHeader('Content-Type', 'text/plain');
  res.end('hello world');
}
var app = connect()
  .use(hello) ③
  .use(logger)
  .listen(3000);
```

- ① always calls `next()` so subsequent middleware are invoked
- ② does not call `next()` since it actually responds to the request
- ③ if the 'hello' middleware goes before 'logger', then 'logger' will never be invoked since 'hello' does not call `next()`

The `hello` middleware would be called first and respond to the HTTP request as expected, however `logger` would never be called because `hello` never calls `next()` at any point, so the control is never passed back to the dispatcher to invoke the next middleware. The moral here is that when a middleware does not call `next()` then any remaining middleware after it in the chain of command will

not be invoked. In this case, placing `hello` in front of `logger` is rather useless, but when leveraged properly the ordering can be used to your benefit.

In summary, when a middleware does not call `next()`, then any remaining middleware after it in the chain of command will not be invoked.

7.3.2 Leveraging middleware order to perform authentication

You can use order of middleware to your advantage like in the case of authentication.

Authentication is a topic relevant to almost any kind of application. Your users need a way to log in, and you need a way to prevent access to content when they are not logged in. Leveraging the order of the middleware lends itself very well to implementing your authentication.

Suppose you have written a middleware called `restrictFileAccess` which grants file access only to valid users. If the user is valid, they are able to continue to the next middleware. If the user is not valid, `next()` is not called. Listing 7.4 shows how the `restrictFileAccess` middleware should go below the `logger` middleware, but before the `serveStaticFiles` middleware.

Listing 7.4 precedence.js: Middleware precedence to restrict file access

```
var connect = require('connect');
connect()
  .use(logger)
  .use(restrictFileAccess)
  .use(serveStaticFiles)
  .use(hello);
```

1 `next()` will only be called if the user is valid

Now that we've discussed middleware precedence and how it's an important tool for constructing application logic, literally the "building blocks" of an app, let's take a look at another feature that Connect provides to help you leverage middleware.

7.4 Mounting middleware & servers

Another feature that Connect provides is the concept of "mounting", a simple yet powerful organizational tool that allows you to define a path prefix that is required in order for the middleware to be called. This concept goes beyond regular middleware use, allowing you to mount entire applications at the specified path. For example, a stand-alone blog application could be mounted at "/blog", though when it is invoked the requested url (`req.url`) will be adjusted so the "/blog" prefix is removed. This allows for seamless reusability with existing node HTTP servers or middleware.

Let's illustrate this functionality with an example. It is common for applications to have their own administration area so that the people in charge can manage the app. Perhaps to moderate comments and approve new users. In our example, this admin area will reside at "/admin" in the application. So now you need a way to make sure that "/admin" is only available to authorized users, and that the rest of the site is available to everybody normally.

"Mounting" is perfect for such a task, because it allows you to apply certain middleware to some parts of your application but not others, based on a given url prefix. Notice in Listing 7.5 how the second and third `use()` calls have the string '`/admin`' as the first argument followed by the middleware itself second. That is the syntax for "mounting" a middleware or server in Connect.

Listing 7.5 mounting.js: The syntax for "mounting" a middleware or server

```
var connect = require('connect');
connect()
  .use(logger)
  .use('/admin', restrict) ①
  .use('/admin', admin)
  .use(hello)
  .listen(3000);
```

- ① when a string is given as the first argument to `.use()`, then Connect will only invoke the middleware when the prefix of "req.url" matches

Armed with that knowledge of how mounting middleware and servers to your application works, let's enhance the "hello world" application you built with an admin area by using mounting and adding two new middleware:

- A `restrict` middleware that ensures a valid user is accessing the page.
- An `admin` middleware which will present the administration area to the user.

7.4.1 A middleware that does authentication

The first middleware you need to add will perform authentication. This will be a generic authentication middleware, not specifically tied to the `"/admin"` `req.url` in any way. However when we mount it onto the application, the middleware will only be invoked when the request url begins with `"/admin"`. This is important because you only want to authenticate users who attempt to access the `"/admin"` url, but still want to pass through regular users as normal.

Listing 7.6 implements crude basic authentication logic. Basic auth¹ is a simple authentication mechanism that uses the HTTP "Authorization" header field with base64 encoded credentials. Once the credentials are decoded by the middleware, then the username and password are checked for correctness. When valid the middleware will simply invoke `next()`, meaning the request is ok to continue processing, otherwise it will throw an error.

Footnote 1 http://wikipedia.org/wiki/Basic_access_authentication

Listing 7.6 restrict.js: A middleware that performs HTTP Basic authentication

```
function restrict(req, res, next) {
  var authorization = req.headers.authorization;
  if (!authorization) return next(new Error('Unauthorized'));
  var parts = authorization.split(' ')
  var scheme = parts[0]
  var auth = new Buffer(parts[1], 'base64').toString().split(':')
  var user = auth[0]
  var pass = auth[1];
  authenticateWithDatabase(user, pass, function (err) { ①
    if (err) return next(err); ②
    next(); ③
  });
}
```

- ① `"authenticateWithDatabase()"` is a hypothetical function that checks the given credentials against a database
- ② Informs dispatcher that an error occurred
- ③ Otherwise call `next()` with no arguments when given valid credentials

Again, notice how this middleware does not do any checking of `req.url` to

ensure that "/admin" is what is actually being requested, this is because Connect is handling this for you. This allows you to write generic middleware. The `restrict` middleware could be used to authenticate another part of the site or another application.

NOTE**Invoking `next` with an Error argument**

Notice in the previous example how the `next` function is invoked with an `Error` object passed in as the argument. When you do this, you are notifying Connect that an application error has occurred. Which means that only "error handling middleware" will be executed for the remainder of this HTTP request. "Error handling middleware" is a topic you will learn about a little later in this chapter. For now, just know that it tells Connect that your middleware has finished and that an error occurred in the process.

When authorization is complete, and no errors have occurred, Connect will continue on to the next middleware, in this case the `admin` middleware.

7.4.2 A middleware that presents an administration panel

The `admin` middleware implements a primitive router using a `switch` statement on the request url. The middleware will present a redirect message when "/" is requested and return a JSON array of usernames when "/users" is requested. The usernames used here are hard-coded for the example, but a real application would more likely grab them from a database.

Listing 7.7 admin.js: Routing admin requests

```
function admin(req, res, next) {
  switch (req.url) {
    case '/':
      res.end('try /users');
      break;
    case '/users':
      res.setHeader('Content-Type', 'application/json');
      res.end(JSON.stringify(['tobi', 'loki', 'jane']));
      break;
  }
}
```

The important thing to note here is that the strings used are "/" and "/users", not "/admin" and "/admin/users". The reason for this is that Connect makes mounted

middleware (and servers) seemingly unaware that they are mounted, treating urls as if they were mounted at "/" all along. This simple technique makes applications and middleware more flexible as they simply do not care "where" they are used.

NOTE

The important thing about mounting

In short, "mounting" allows you to write middleware from the root level (again the "/" base `req.url`) or reuse an existing server on any arbitrary path prefix, without altering the code each time. So when a middleware or server is mounted at "/blog", it can still be written using and "/article/1", instead of "/blog/article/1". This separation of concerns means you can reuse the blog server in multiple places while never needing to alter the code.

For example, mounting would allow a "blog" application to be hosted at `http://foo.com/blog` as well as `http://bar.com/posts`, without any change to the blog app code accounting for the change in url. This is because Connect alters the `req.url` by stripping off the prefix portion when mounted. The end result is that the blog app can be written with paths relative to "/", and doesn't even need to know about "/blog", or "/posts". The requests will follow the same middleware and share the same state. Consider the server setup used here, which reuses the hypothetical "blog" application by mounting it at two different mount points:

```
var connect = require('connect');
connect()
  .use(logger)
  .use('/blog', blog)
  .use('/posts', blog)
  .use(hello)
  .listen(3000);
```

TESTING IT ALL OUT

Now that the middleware are taken care of it is time to take your application for a test drive using curl. You can see that regular urls other than "/admin" will invoke the `hello` middleware as expected:

```
$ curl http://localhost
hello world
$ curl http://localhost/foo
```

```
hello world
```

You can also see that the `restrict` middleware will return an error to the user when no credentials are given or incorrect credentials are used:

```
$ curl http://localhost/admin/users
Error: Unauthorized
  at Object.restrict [as handle]
  (E:\transfer\manning\node.js\src\ch7\multiple_connect.js:24:35)
  at next
  (E:\transfer\manning\node.js\src\ch7\node_modules\
  ↵connect\lib\proto.js:190:15)
  ...
$ curl --user jane:ferret http://localhost/admin/users
Error: Unauthorized
  at Object.restrict [as handle]
  (E:\transfer\manning\node.js\src\ch7\multiple_connect.js:24:35)
  at next
  (E:\transfer\manning\node.js\src\ch7\node_modules\
  ↵connect\lib\proto.js:190:15)
  ...
```

And finally you can see that only when authenticated as "tobi" will the admin middleware be invoked and the server responds with the JSON array of users:

```
$ curl --user tobi:ferret http://localhost/admin/users
[ "tobi", "loki", "jane" ]
```

See how simple yet powerful mounting is? Now, let's take a look at some techniques for creating configurable middleware.

7.5 Creating configurable middleware

You have learned some of the middleware basics, but now we'll go into detail showing you how to create more generic and reusable middleware so that you can reuse them in your applications with only some additional configuration required, rather than re-implementing a middleware from scratch with your application-specific needs. These are tasks like configurable logging, routing requests to callback functions, rewriting urls and an infinite amount more. You will be able to reuse the simple middleware you write in this section in apps you design in the future. Reusability is one of the major benefits of writing middleware.

Middleware commonly follow a simple convention in order to provide configuration capabilities to developers: using a function that returns another function (this is a powerful JavaScript feature, typically called a "closure"). The basic structure for configurable middleware of this kind looks like:

```
function setup(options) {
  // setup logic
  ①
  return function(req, res, next) {
    // middleware logic
    ②
  }
}
```

- ① addition middleware initialization here
- ② options still accessible even though outer function has returned

Which is used like:

```
app.use(setup({some: 'options'}))
```

Notice that the `setup` function is invoked in the `app.use` line where in our previous examples we were just passing a reference to the function.

In this section we will apply this technique to build three reusable "configurable" middleware:

- A logger middleware with configurable printing format
- A router middleware that invokes functions based on the requested url
- A url rewriter middleware that converts URL "slugs" to ids

7.5.1 Example 1: Creating a configurable logger middleware

The logger middleware you created before was *not* configurable. It was hard-coded to print out the requests' `req.method` and `req.url` when invoked. But what about when you want to change what the logger displays at some point in the future? While you could modify your logger middleware manually, a better solution would be to make the logger configurable from the start instead of hard-coding the values. So let's do that.

In practice, using a "configurable" middleware is just like using any of the

middleware you have created so far, only now you can pass additional arguments to the middleware to alter its behavior. Using the middleware in your application might look a little like the following example, where the logger can accept a string that describes the format that the logger should print out:

```
var app = connect()
  .use(logger(':method :url'))
  .use(hello);
```

To implement the configurable logger middleware you first define a `setup` function that accepts a single argument expected to be a string (in this example we will name it `format`). When invoked, a function is returned which is the actual middleware Connect will use. The returned middleware retains access to the `format` variable, even after the `setup` function has returned, since it is defined within the same JavaScript "closure". The logger then replaces the tokens in the `format` string with the associated request properties on the `req` object, then logs to stdout and calls `next()` as shown in listing 7.8.

Listing 7.8 logger.js: A configurable "logger" middleware for Connect

```
function setup(format) { ①
  var regexp = /:(\w+)/g; ②
  return function logger(req, res, next) { ③
    var str = format.replace(regexp, function(match, property){ ④
      return req[property];
    });
    console.log(str); ⑤
    next(); ⑥
  }
} ⑦
module.exports = setup;
```

- ① The setup function may be called multiple times with different configurations
- ② The logger middleware will use a RegExp to match the request properties
- ③ Here is the actual logger middleware that Connect will use
- ④ Use the regexp to format the log entry for the request
- ⑤ Print out the request log entry to the console
- ⑥ Finally, pass control on to the next middleware
- ⑦ Directly export the logger setup function

Since we have now created this logger middleware as a configurable middleware, you can `.use()` the logger multiple times in a single application with different configurations, or re-use this logger code in any number of future applications you might develop that accept middleware. This simple concept of "configurable" middleware is used throughout the Connect community, and is used for all core Connect middleware, even those that are not configurable to maintain consistency. Now let's write a middleware that requires a little bit more involved logic. Let's create a router to map incoming requests to business logic!

7.5.2 Example 2: Building a routing middleware

Routing is a crucial web application concept. Put simply, it's a method of mapping incoming request URLs to a function that employs business logic. Routing comes in many shapes and sizes, ranging from highly abstract controllers used by frameworks like Ruby on Rails, or simpler, less abstract HTTP method and path based routing provided by frameworks like Express and Ruby's Sinatra.

Using a simple router in your application might look something like listing 7.9, where HTTP verbs and paths are represented by a simple object and some callback functions, and string tokens prefixed with ":" represent a path segment that accept user-input, matching paths like `/user/12`. The result is an application with a collection of handler functions that will be invoked when the request method and URL match one that has been defined.

Listing 7.9 connect-router-usage.js: What usage of the router middleware will look like

```

var connect = require('connect');
var router = require('./middleware/router');      1
var routes = {          2
  GET: {
    '/users': function(req, res){
      res.end('tobi, loki, ferret');
    },
    '/user/:id': function(req, res, id){          3
      res.end('user ' + id);
    }
  },
  DELETE: {
    '/user/:id': function(req, res, id){
      res.end('deleted user ' + id);
    }
  }
};
connect()
  .use(router(routes))      4
  .listen(3000);

```

- ① the 'router' middleware will be defined later in this section
- ② to use the 'router', you create an object whose keys are the request methods and values are objects containing the route URLs as keys and 'route' functions as values.
- ③ each entry maps to a request URL and contains a callback function to invoke
- ④ you pass the "routes" object to the 'router' setup function

Since there are no restrictions on the number of middleware or number of times a middleware may be used, it's fully possible to define several routers in a single application. This could be useful to do for organizational purposes. Suppose we have user related routes and some administration routes. You could separate these into module files and require them for the router middleware as shown in the following snippet.

```

var connect = require('connect');
var router = require('./middleware/router');
connect()
  .use(router(require('./routes/user')))
  .use(router(require('./routes/admin')))
  .listen(3000);

```

Now let's actually build this router middleware! This will be the most complicated middleware example we have gone over so far, so let's quickly run through the logic this router will implement in a flowchart, shown in figure 7.2.

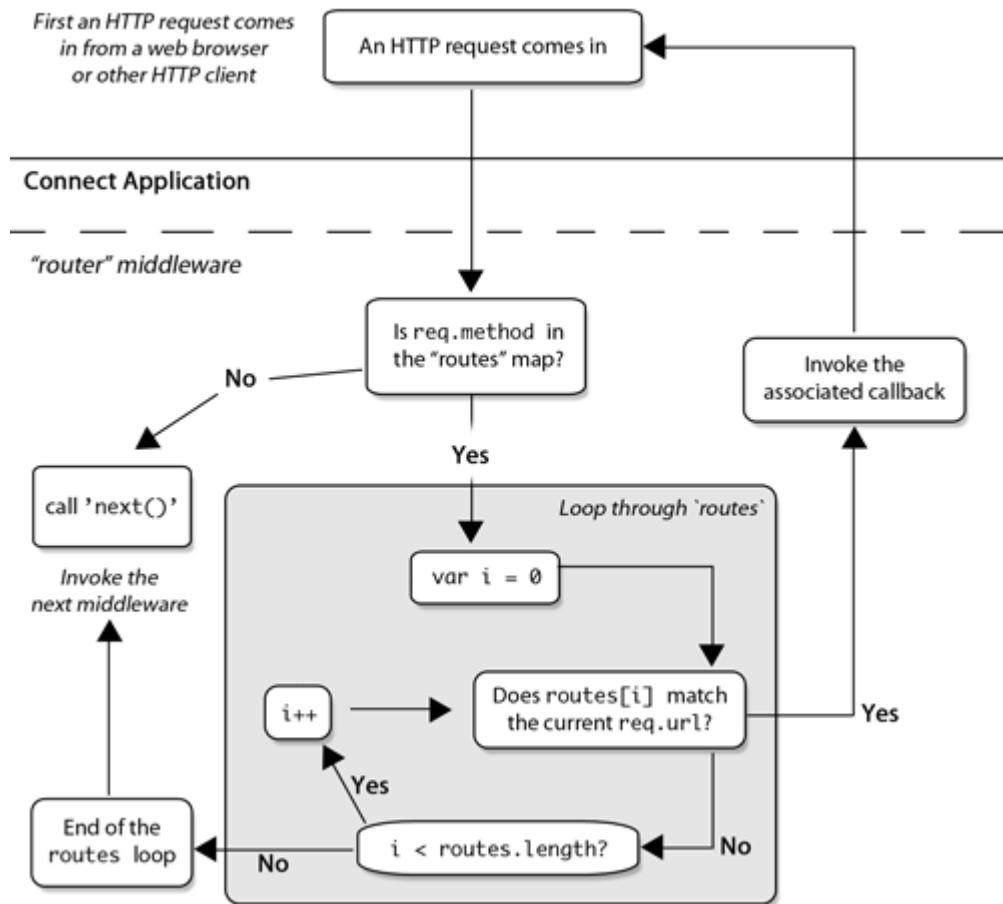


Figure 7.2 Flowchart of the "router" middleware's logic

You can see how the flowchart almost acts as "pseudo code" for our middleware, which helps us implement the actual code for the router. See the middleware in its entirety in listing 7.10.

Listing 7.10 connect-router.js: Simple routing middleware

```

var parse = require('url').parse;
module.exports = function route(obj) {
  return function(req, res, next){
    if (!obj[req.method]) { ①
      next();
      return;
    }
    var routes = obj[req.method] ③
    var url = parse(req.url) ④
    var paths = Object.keys(routes) ⑤
    for (var i = 0; i < paths.length; i++) { ⑥
      var path = paths[i];
      var fn = routes[path];
      path = path
        .replace(/\/\/g, '\\\\/')
        .replace('/:(\w+)/g, '([^\\\/]+)');
      var re = new RegExp('^' + path + '$'); ⑦
      var captures = url.pathname.match(re)
      if (captures) { ⑧
        var args = [req, res].concat(captures.slice(1));
        fn.apply(null, args);
        return; ⑩
      }
    }
    next();
  };
};

```

- ① first check to make sure the req.method is defined
- ② if not, then invoke next() and stop and further execution
- ③ object keyed with the paths
- ④ parse url for matching against the pathname
- ⑤ the paths for this http method as an array
- ⑥ loop the paths
- ⑦ construct the regular expression
- ⑧ attempt match against the pathname
- ⑨ pass the capture groups
- ⑩ return when a match is found to prevent the next() call below

This "router" makes a great example of configurable middleware, as it follows the traditional format of having a "setup" function return a real middleware for Connect applications to use. In this case, it accepts a single argument, the "routes"

object, which contains the map of HTTP verbs, request urls, and callback functions. It first checks to see if the current `req.method` is defined in the "routes" map, and stops further processing in the router if not (by invoking `next()`). After that it loops through the defined paths and checks to see if one matches the current `req.url`. If it finds a match, then the match's associated callback function will be invoked, hopefully completing the HTTP request.

This is a complete middleware with a couple nice features, but you could easily expand on this router middleware. For example, you could utilize the power of closures to cache the regular expressions which would otherwise be compiled per-request.

Another great use of middleware is to rewrite urls. Next up we'll take a look at rewriting urls to accept blog post "slugs" rather than showing post ids in the url.

7.5.3 Example 3: Building a middleware to rewrite urls

Rewriting URLs can be very helpful. Suppose you want to accept a request to "/blog/posts/my-post-title", lookup the post id based on the ending post title portion (commonly known as the "slug" part of the URL), and then transform the URL to "/blog/posts/<post-id>". This is a perfect task for middleware!

The small blog application in the following snippet consists of the following middleware setup, where we first rewrite the url based on the "slug" with a `rewrite` middleware, and then pass control to the `showPost` middleware:

```
var connect = require('connect')
var url = require('url')
var app = connect()
  .use(rewrite)
  .use(showPost)
  .listen(3000)
```

The following `rewrite` middleware implementation in listing 7.11 parses the url to access the pathname, then matches with a regular expression. The first capture group (the slug) is passed to a hypothetical `findPostIdBySlug` function that looks up the blog post id by slug. When successful you can then re-assign the request url (`req.url`) to whatever you like. In this case the id is appended to "/blog/post/" so that the subsequent middleware can perform the blog post lookup via id.

Listing 7.11 connect-rewrite.js: Middleware that rewrites the request url based on a slug name.

```

var path = url.parse(req.url).pathname;
function rewrite(req, res, next) {
  var match = path.match(/^\/blog\/posts\/(.+)/)
  if (match) { ①
    findPostIdBySlug(match[1], function(err, id) {
      if (err) return next(err); ②
      if (!id) return next(new Error('User not found')); ③
      req.url = '/blog/posts/' + id; ④
      next();
    });
  } else {
    next();
  }
}

```

- ① Only perform the lookup on /blog/posts requests
- ② If there was an error with the lookup, then inform the error handler and stop processing
- ③ If there was no matching id for the "slug" name, call next() with a "User not found" Error argument
- ④ Finally overwrite the req.url property so that subsequent middleware can utilize the real id

NOTE

What these examples demonstrate

The important takeaway from these examples is that you should think "small and configurable pieces" when building your middleware. That is, build lots of tiny, modular and reusable middleware, that as a sum make up your entire application. Keeping your middleware small and focused really helps with breaking down complicated application logic into smaller pieces.

Next up let's take a look at the final middleware concept that Connect provides, this time specifically for handing application errors.

7.6 Error handling middleware

Let's face it, all applications have errors, whether at system level or user level, and being well prepared for those situations you aren't anticipating is a smart thing to do. Connect implements an error-handling variant of middleware, following the same rules as "regular" middleware however accepting an error object along with the request and response objects. Error handling in any kind of application is an opinionated subject that should be left up to the application itself to decide how to handle, and not the framework, and that is exactly the goal of error-handling middleware.

Connect error handling is intentionally minimal allowing the developer to specify how errors should be handled. For example, you could pass only system/application errors through the middleware (e.g. "foo is undefined") or user errors (e.g. "password is invalid") or a combination of both. Connect leaves the opinions up to you on which is best for your application. In this section we will make use of both types and you will learn all about how error-handling middleware work and useful patterns that can be applied while using them:

- Connect's default error-handler
- Handing application errors yourself
- Using multiple error-handling middleware

7.6.1 Connect's default error-handler

Consider the following middleware which will throw a `ReferenceError` error due to the function `foo()` not being defined by the application.

```
var connect = require('connect')
connect()
  .use(function hello(req, res) {
    foo();
    res.setHeader('Content-Type', 'text/plain');
    res.end('hello world');
  })
  .listen(3000)
```

By default, Connect will respond with a 500 status code and a response body containing the text "Internal Server Error", and more information about the error itself. This is fine, but in any kind of real application would like to do more specialized things with those errors, like send them off to a logging deamon.

7.6.2 Handling application errors yourself

Connect also offers a way for you to handle application errors yourself using error-handling middleware. For instance, in development you might want to respond with a JSON representation of the error for quick and easy reporting to the client-side, whereas you'd want to respond with a simple "Server error" in production so as not to expose sensitive internal information about the application that an attacker could leverage, such stack traces, file names and line numbers.

NOTE

Use NODE_ENV to set the application's "mode"

A common Connect convention is to use the "NODE_ENV" environment variable (`process.env.NODE_ENV`) to toggle the behavior between different server environments, like "production" and "development".

To define an error handling middleware the function must be defined to accept four arguments (`err, req, res, next`) as shown in the listing 7.12, whereas regular middleware take the form (`req, res, next`) as you have already learned.

Listing 7.12 errorHandler.js: Error handling middleware in Connect

```
function errorHandler() {
  var env = process.env.NODE_ENV || 'development';
  return function(err, req, res, next) { ①
    res.statusCode = 500;
    switch (env) { ②
      case 'development':
        res.setHeader('Content-Type', 'application/json');
        res.end(JSON.stringify(err));
        break;
      default:
        res.end('Server error');
    }
  }
}
```

- ① Error handling middleware define 4 arguments
- ② This example errorHandler behaves differently depending on the "env"

When Connect encounters an error it will switch to invoking only error handling middleware as you can see in figure 7.3.

Lifecycle of an HTTP Request causing an error in a Connect application

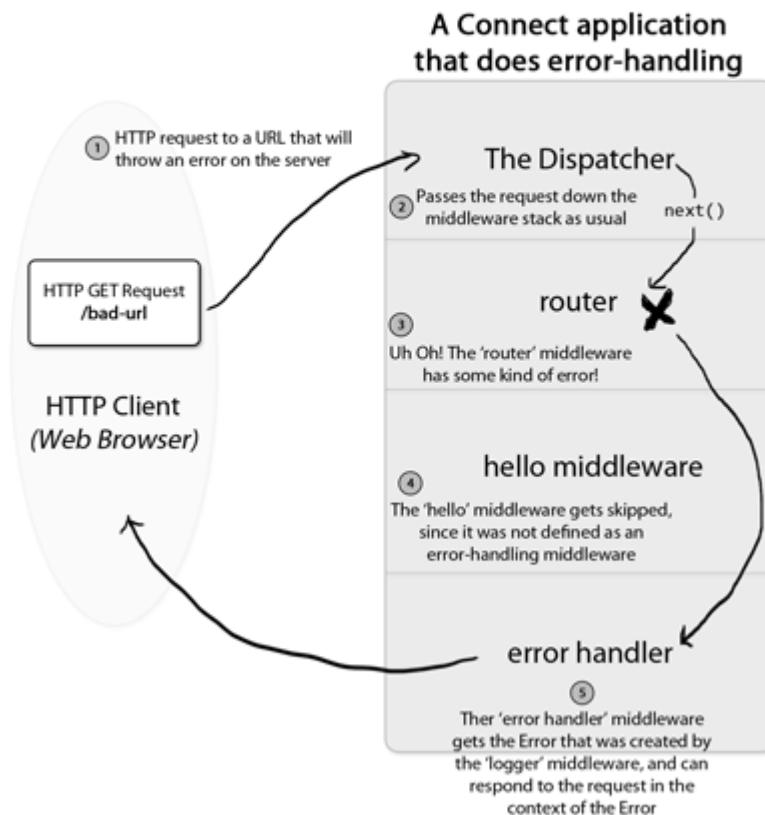


Figure 7.3 The lifecycle of an HTTP request causing an error in a Connect server

For example, if the first routing middleware for the "user" routes caused an error, both the "blog" and "admin" middleware would be skipped, as they do not act as error handling middleware since they only define 3 arguments. Connect would then see that `errorHandler` accepts the `error` argument, and invoke it.

```
connect()
  .use(router(require('./routes/user')))
  .use(router(require('./routes/blog'))) // Skipped
  .use(router(require('./routes/admin'))) // Skipped
  .use(errorHandler());
```

7.6.3 Using multiple error-handling middleware

Using a variant of middleware for error handling can be useful for separating error handling concerns. Suppose your app has a web service mounted at "/api". You may wish that the web application errors render an html error page to the user, but any "/api" requests return more verbose errors, perhaps always responding with JSON so that receiving clients can easily parse the errors and react properly.

To illustrate this "/api" scenario, consider the following application, where `app` is the main web application, and `api` is mounted to "/api". Follow and implement this small example as we go along.

```
var api = connect()
  .use(users)
  .use(pets)
  .use(errorHandler);
var app = connect()
  .use(hello)
  .use('/api', api)
  .use(errorPage)
  .listen(3000);
```

This setup created by the configuration for this application is easily visualized in figure 7.4:

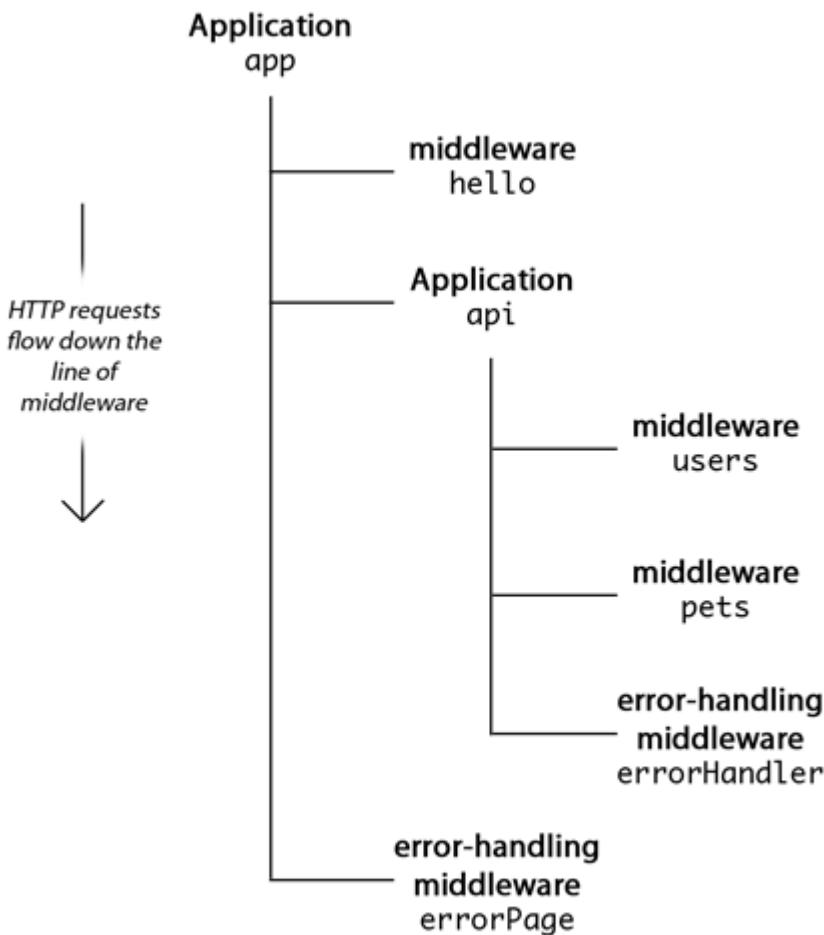


Figure 7.4 Layout of the example application with two error-handling middleware

So now we need to implement each of the application middleware:

- The "hello" middleware which responds with "Hello World\n"
- The "users" middleware will throw a "notFound" Error when a user does not exist
- The "pets" middleware will cause a ReferenceError to be thrown, to demonstrate the error handler
- The "errorHandler" middleware handles any errors from the `api` app
- The "errorPage" middleware which will handle any error from the main `app` app

IMPLEMENTING THE "HELLO" MIDDLEWARE

The implementation of the `hello` middleware is simply a function that matches `"/hello"` with a regular expression as shown in the following snippet:

```
function hello(req, res, next) {
```

```

if (req.url.match(/^\hello/)) {
  res.end('Hello World\n');
} else {
  next();
}
}

```

In this case, there is no possible way for an error to occur in such a simple middleware.

IMPLEMENTING THE "USERS" MIDDLEWARE

The `users` middleware is slightly more complex. Implemented in listing XREF `connect-users`, we match the `req.url` using a regular expression, then check if the user index exists by using `match[1]` which is the first capture group for our match. If the user exists, then is serialized as JSON, otherwise an error is passed to the `next()` function with its `notFound` property set to "true", allowing you to unify error handling logic in the error handling middleware later.

Listing 7.13 user_middleware.js: Example "users" middleware that searches for a user in "db" and throws a "notFound" error when that fails

```

var db = {
  users: [
    { name: 'tobi' },
    { name: 'loki' },
    { name: 'jane' }
  ]
};
function users(req, res, next) {
  var match = req.url.match(/^\user\/(.+)/)
  if (match) {
    var user = db.users[match[1]];
    if (user) {
      res.setHeader('Content-Type', 'application/json');
      res.end(JSON.stringify(user));
    } else {
      var err = new Error('User not found');
      err.notFound = true;
      next(err);
    }
  } else {
    next();
  }
}

```

IMPLEMENTING THE "PETS" MIDDLEWARE

The following is the partially implemented `pets` middleware. To illustrate how you can apply logic to the errors based on properties such as the `err.notFound` boolean you assigned in the previous middleware. Here the undefined `foo()` function will trigger an exception, which will not have this property.

```
function pets(req, res, next) {
  if (req.url.match(/^\pet\/(.+)/)) {
    foo();
  } else {
    next();
  }
}
```

IMPLEMENTING THE "ERRORHANDLER" MIDDLEWARE

Finally, you're at the `errorHandler` middleware! Contextual error messages are especially important for web services to provide appropriate feedback to the consumer, without giving away too much information! You certainly do not want to expose errors such as "`{"error": "foo is not defined"}`", or even worse, full stack traces because an attacker could use this information against you. Therefore you should only respond with error messages that you know are safe, as the following `errorHandler` implementation does in listing 7.14.

Listing 7.14 `error_handling.js`: A production-ready error handling middleware which doesn't expose too much information

```
function errorHandler(err, req, res, next) {
  console.error(err.stack);
  res.setHeader('Content-Type', 'application/json');
  if (err.notFound) {
    res.statusCode = 404;
    res.end(JSON.stringify({ error: err.message }));
  } else {
    res.statusCode = 500;
    res.end(JSON.stringify({ error: 'Internal Server Error' }));
  }
}
```

This error handling middleware uses the `err.notFound` property set earlier to distinguish between server errors and client errors. Another approach would be

to check if the error is an `instanceof` some other kind of error (such as a `ValidationError` from some validation module) and respond accordingly. If the server were to accept an HTTP request to, say, "/user/ronald", which does not exist in our database, then the `users` middleware from earlier would throw a "notFound" error, and when it got to the `errorHandler` middleware it would trigger the `err.notFound` code path, which returns a 404 status code along with the `err.message` property as a JSON object. Figure 7.5 shows how the raw output looks in a web browser.



Figure 7.5 The JSON object output of the "User not found" error

IMPLEMENTING THE "ERRORPAGE" MIDDLEWARE

The `errorPage` middleware is the second error-handling middleware in our example application. Because the previous error-handling middleware never calls `next(err)`, that leaves the only opportunity for this middleware to be invoked is by an error occurring in the `hello` middleware. However, that middleware is very unlikely to generate an error, leaving very little chance for this `errorPage` middleware ever to be invoked. That said, we will leave implementing this second error-handling middleware up to you, as it is literally optional in this example.

Ok so finally your application is ready. You can fire up the server which we set to listen on port 3000 back in the beginning. You can play around with it using a browser or `curl` or any other HTTP client. Try triggering the various routes of the error-handler by requesting an invalid user, or requesting one of the "pets" entries.

To re-emphasize, error handling is a *crucial* aspect of any kind of application that should not go left unaccounted for. Error handling middleware are a clean way to unify the error-handling logic in your application in a centralized location, and you should always include at least one error handling middleware in your application; at least by the time it hits production.

7.7 Summary

In this chapter you've learned everything you need to know about the small but powerful Connect framework. You learned about how the dispatcher works and how to build middleware to make your applications modular and flexible. You learned how to mount middleware to a particular base URL which enables you to create applications inside of applications. You also were exposed to configurable middleware that take in settings and thus can be repurposed and tweaked. Lastly you learned how to handle errors that occur within middleware.

Now that the fundamentals are out of the way, you can learn about the middleware that Connect provides out of the box in the next chapter.

A large, light gray, stylized number '8' is positioned in the upper right corner of the page.

Connect's built-in middleware

In this chapter:

- Middleware for parsing cookies, request bodies and query-strings
- Middleware that implement core web application needs
- Middleware that handle web application security
- Middleware for serving static files

In the previous chapter you learned what middleware are, how to create them, and how to use them with Connect. But the middleware functionality of Connect is only the base of the framework. The real power of Connect comes from its bundled middleware. These middleware solve many common web application needs that we have been talking about, such as session management, cookie parsing, body parsing, request logging and much more.

Connect provides a variety of useful built-in middleware, ranging in complexity which provide a great starting point for building simple web servers, or for use in higher level web frameworks. Throughout this chapter we'll provide an overview of the more commonly used bundled middleware that you may use in your own application with an explanation and simple example. Table 8.1 gives an overview of the middleware that will be covered for quick reference:

Table 8.1 Connect middleware quick reference guide

Middleware Name	Section #	Summary
cookieParser()	8.1.1	Provides <code>req.cookies</code> and <code>req.signedCookies</code> for subsequent middleware to use.
bodyParser()	8.1.2	Provides <code>req.body</code> and <code>req.files</code> for subsequent middleware to use.
limit()	8.1.3	Restricts request body sizes based on a given byte length limit. Must go before the <code>bodyParser</code> middleware.
query()	8.1.4	Provides <code>req.query</code> for subsequent middleware to use.
logger()	8.2.1	Logs configurable information about incoming HTTP requests to a Stream, like <code>stdout</code> or a log file.
favicon()	8.2.2	Responds to the <code>/favicon.ico</code> HTTP requests. Usually placed before the <code>logger</code> middleware, so that you don't have to see it in your log files.
methodOverride()	8.2.3	Allows you to fake <code>req.method</code> for browsers which are incapable of using the proper method. Depends on <code>bodyParser</code> .
vhost()	8.2.4	Routes to a given middleware and/or HTTP Server instances based on a specified hostname (i.e. "nodejs.org").
session()	8.2.5	Sets up an HTTP session for a user and provides a persistent <code>req.session</code> object in between requests. Depends on <code>cookieParser</code> .
basicAuth()	8.3.1	Provides HTTP Basic Authentication for your application.

<code>csrf()</code>	8.3.2	Protects against "Cross-site request forgery" attacks in HTTP forms. Depends on <code>session</code> .
<code>errorHandler()</code>	8.3.3	A middleware useful for development that returns stack traces to the client when a server-side error occurs. <i>Do not use for production!</i>
<code>static()</code>	8.4.1	Serves files from a given directory to HTTP clients. Works really well with Connect's "mounting" feature.
<code>compress()</code>	8.4.2	Optimizes HTTP responses using gzip compression.
<code>directory()</code>	8.4.3	Serves directory listings to HTTP clients, providing the optimal result based on the client's <code>Accept</code> request header (plain text, JSON, or HTML).

First up we'll look at the middleware that implement various parsers needed to build proper web applications, since these are the foundation for most of the other middleware.

8.1 Middleware for parsing cookies, request bodies and query-strings

Node core doesn't provide modules for higher level web application concepts like parsing cookies, buffering request bodies, or parsing complex query-strings, so Connect provides those out of the box for your application to use. In this section we will cover the 4 built-in middleware that deal with parsing input request data:

- `cookieParser` - for parsing "cookies" from web browsers into `req.cookies`
- `bodyParser` - consumes and parses the request body into `req.body`
- `limit` - goes hand in hand with `bodyParser` to limit requests from getting too big
- `query` - parses the request url query-string into `req.query`

Let's start off covering cookies, often used by web browsers to simulate "state" since HTTP is a stateless protocol.

8.1.1 cookieParser: Parsing HTTP cookies

Connect's cookie parser supports "regular" cookies, signed cookies, and special "JSON cookies" out of the box. By default, regular unsigned cookies are used, populating the `req.cookies` object. But if you want signed cookie support, which are required by the `session()` middleware, then you'll want to pass a "secret" string when creating the `cookieParser()` instance.

NOTE

Setting cookies on the server-side

The `cookieParser()` middleware doesn't provide any helpers for setting outgoing cookies. For this, you should use the `res.setHeader()` function with "Set-Cookie" as the header name. Connect patches Node's default `res.setHeader()` function to special-case the "Set-Cookie" headers, so that it "just works" as you would expect it to.

BASIC USAGE

The "secret" passed as the argument is used to sign and unsign the cookies, allowing Connect to determine if the contents have been tampered with (as only your application knows the secret's value). Typically the secret should be a reasonably large string, potentially randomly generated. Here the secret will be "tobi is a cool ferret":

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

var app = connect()
  .use(connect.cookieParser('tobi is a cool ferret'))
  .use(function(req, res){
    console.log(req.cookies);
    console.log(req.signedCookies);
    res.end('hello\n');
  }).listen(3000);
```

The `req.cookies` and `req.signedCookies` properties get set to Objects representing the parsed "Cookie" header that got sent with the request. If there were no cookies sent with the request, than they will both be empty objects.

REGULAR COOKIES

If you were to fire some HTTP requests off to this server using `curl(1)` without the "Cookie" header field, then you'll see that both of the `console.log()` calls output an empty object:

```
$ curl http://localhost:3000/
{}
{}
```

Now try sending a few cookies. You'll see that both are available as properties of `req.cookies`:

```
$ curl http://localhost:3000/ -H "Cookie: foo=bar, bar=baz"
{ foo: 'bar', bar: 'baz' }
{}
```

SIGNED COOKIES

Signed cookies are better suited for sensitive data as the integrity of the cookie data can be verified, helping to prevent "man-in-the-middle" attacks¹. These cookies are placed in the `req.signedCookies` object when valid. The reasoning behind two separate objects is that it shows the developers intention. If we were to place both signed and unsigned cookies in the same object a regular cookie could be crafted to contain data intended as a signed cookie.

Footnote 1 http://en.wikipedia.org/wiki/Man-in-the-middle_attack

A signed cookie looks something like "tobi.DDm3AcVxE9oneYnbmpqxooyhyKsk", where the left-hand side of the "." cookie's value and the right-hand is the secret hash generated on the server with sha1 HMAC (Hash-based Message Authentication Code). When Connect attempts to unsign the cookie it will fail if either the value or HMAC has been altered.

Suppose for example you set a signed cookie with the key "name" and the value "luna". `cookieParser` would encode the resulting cookie to "luna.PQLM0wNvqOQEObZXUkWbS5m6Wlg". The hash portion is checked on each request and when the cookie is sent intact, it will be available as `req.signedCookies.name`:

```
$ curl http://localhost:3000/ -H "Cookie:
➡ name=luna.PQLM0wNvqOQEObZXUkWbS5m6Wlg"
{}
{ name: 'luna' }
GET / 200 4ms
```

Now if the cookie's value were to change as shown in the next curl command,

the "name" cookie will be available as `req.cookies.name`, as it was not valid. However it may still be of use for debugging or application-specific purposes.

```
$ curl http://localhost:3000/ -H "Cookie:
  name=manny.PQLM0wNvqOQEObZXUkWbS5m6Wlg"
{ name: 'manny.PQLM0wNvqOQEObZXUkWbS5m6Wlg' }
{}
GET / 200 1ms
```

JSON COOKIES

The special JSON cookie is a cookie prefixed with "j:", and works with both signed and unsigned cookie variants, notifying Connect that it is intended to be serialized JSON. Frameworks such as Express can use this functionality to provide a more intuitive cookie interface instead of requiring that developers manually serialize and parse the JSON cookie values.

```
$ curl http://localhost:3000/ -H 'Cookie: foo=bar, bar=j:{ "foo": "bar" }'
{ foo: 'bar', bar: { foo: 'bar' } }
{}
GET / 200 1ms
```

As mentioned, JSON cookies work when signed as illustrated by the following request:

```
$ curl http://localhost:3000/ -H "Cookie:
  cart=j:[{"items": [1]}.sD5p6xFFBO/4ketA1OP43bcjs3Y"
{}
{ cart: { items: [ 1 ] } }
GET / 200 1ms
```

SETTING OUTGOING COOKIES

As noted before, the `cookieParser()` middleware doesn't provide any functionality for writing outgoing headers to the HTTP client via the "Set-Cookie" header. Connect does, however, patch in explicit support for multiple "Set-Cookie" headers via the `res.setHeader()` function.

So say you wanted to set a cookie named "foo" with the string value "bar", Connect makes this one line of code by calling `res.setHeader()`. You can also optionally set the various options of a cookie, like its expiration date or having it be "HttpOnly", like shown in the second `setHeader()` call here:

```

var connect = require('connect');

var app = connect()
  .use(function(req, res){
    res.setHeader('Set-Cookie', 'foo=bar');
    res.setHeader('Set-Cookie', 'tobi=ferret';
      ↗ Expires=Tue, 08 Jun 2021 10:18:14 GMT');
    res.end();
  }).listen(3000);

```

Now if you check out the headers that this server sends back to the HTTP request by using the `--head` flag of curl, then you can see the "Set-Cookie" headers as you would expect:

```

$ curl http://localhost:3000/ --head
HTTP/1.1 200 OK
Set-Cookie: foo=bar
Set-Cookie: tobi=ferret; Expires=Tue, 08 Jun 2021 10:18:14 GMT
Connection: keep-alive

```

And that's all there is to sending cookies with your HTTP response. You can store any kind of text data in cookies, but it has become ubiquitous to store a single "session cookie" on the client-side so that you can have full user-state on the server. This "session" technique is encapsulated in the `session()` middleware which you will learn about a little later in this chapter.

Another extremely common need in web application development is parsing incoming request bodies. Next we'll look at the `bodyParser()` middleware and how it will make your life as a Node developer easier.

8.1.2 bodyParser: Parsing request bodies

A very common need for all kinds of web applications is accepting input from the user. Let's say you wanted to accept user file uploads using the `<input type="file">` HTML tag. One line of code adding the `bodyParser()` middleware is all it takes. This is an extremely helpful middleware, which is actually an aggregate of three other smaller middleware: namely `json()`, `urlencoded()` and `multipart()`. A unified `req.body` property is provided by parsing JSON, x-www-form-urlencoded, and multipart/form-data requests. When the request is a multipart/form-data request, like a file upload, the `req.files` object will also be available.

BASIC USAGE

Let's try it out! Suppose you want to accept registration information for your application through a JSON request. All you have to do is add the `bodyParser()` middleware before any other middleware that will access the `req.body` object. Optionally you may pass in an options object which will get passed through to the sub-middleware mentioned before:

```
var app = connect()
  .use(connect.bodyParser())
  .use(function(req, res){
    // .. do stuff to register the user ..
    res.end('Registered new user: ' + req.body.username);
  });

```

PARSING JSON DATA

The following `curl(1)` request could be used to submit data to your application, sending a JSON object with the "username" property set to "tobi":

```
$ curl -d '{"username":"tobi"}' -H "Content-Type: application/json"
→ http://localhost
Registered new user: tobi
```

PARSING REGULAR <FORM> DATA

Because `bodyParser()` parses based on the Content-Type, the input format is abstracted away from the application, so that all you need to care about is the resulting `req.body` data object. For example, the following `curl(1)` command will send x-www-form-urlencoded data, but the middleware will work as expected without any additional change to the code, and will provide the `req.body.name` property just as before.

```
$ curl -d name=tobi http://localhost
Registered new user: tobi
```

PARSING MULTIPART <FORM> DATA

The `bodyParser` parses multipart/form-data, typical for file uploads, is backed by the 3rd-party module "formidable"², the same one discussed in Chapter 4. To test this functionality out you can log both the `req.body` and `req.files` objects to inspect them.

Footnote 2 <https://github.com/felixge/node-formidable>

```
var app = connect()
  .use(connect.bodyParser())
  .use(function(req, res){
    console.log(req.body);
    console.log(req.files);
    res.end('thanks!');
  });
}
```

Now using `curl(1)` you can simulate a browser file upload using the `-F` or `--form` flag which expects the name of the field and the value. This request will upload a single image on disk named "photo.png", as well as the field "name" containing "tobi":

```
$ curl -F image=@photo.png -F name=tobi http://localhost
thanks!
```

If you take a look at the output of the application, you'll see something very similar to the following example output, where the first object represents `req.body`, and the second is `req.files`. As you can see in the output `req.files.image.path` would be available to your application to rename the file on disk, transfer the data to a worker for processing, upload to a CDN, or anything else your app requires.

```
{ name: 'tobi' }
{ image:
  { size: 4,
    path: '/tmp/95cd49f7ea6b909250abbd08ea954093',
    name: 'photo.png',
    type: 'application/octet-stream',
    lastModifiedDate: Sun, 11 Dec 2011 20:52:20 GMT,
    length: [Getter],
    filename: [Getter],
    mime: [Getter] } }
```

Now that we've detailed the body parsers, you may be wondering "if `bodyParser()` buffers the json and x-www-form-urlencoded request bodies in memory producing one large string, then couldn't an attacker produce extremely large bodies of JSON to deny service to valid visitors?" The answer to that is essentially yes, and because of this the `limit()` middleware exists, allowing you to customize what an acceptable request body size is. Let's take a look.

8.1.3 *limit: Request body limiting*

Simply parsing request bodies is not enough. Developers also need to properly classify acceptable requests and place limits when appropriate. The `limit()` middleware is designed to help prevent huge requests whether they are intended to be malicious or not. For example, an innocent user uploading a photo may accidentally upload an uncompressed "RAW" image, commonly saved by SLR digital cameras and consisting of several hundred megabytes, or a malicious user may craft a massive JSON string to lock up `bodyParser()`, and in turn V8's `JSON.parse()` method. You must take care to configure your server to handle these situations.

WHY IS LIMIT() NEEDED?

Let's take a look at how a malicious user can render a vulnerable server useless. First, create the following small Connect application named "server.js", which does nothing other than parse request bodies using the `bodyParser()` middleware.

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

var app = connect()
  .use(connect.bodyParser());

app.listen(3000);
```

Now create a file name "dos.js", as shown in listing 8.1. You can see how a malicious user could use Node's HTTP client to attack the HTTP server from before, simply by writing several Megabytes of JSON data.

Listing 8.1 dos.js: Perform a Denial of Service attack on a vulnerable HTTP server

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

var req = http.request({
  method: 'POST',
  port: 3000,
  headers: {
    'Content-Type': 'application/json'
  }
});

req.write('[');
var n = 300000;
while (n--) {
  req.write('"foo",');
}
req.write(' "bar" ]');

req.end();
```

1 notify the server
that we are sending
JSON data

2 begin sending a
very large Array
object

3 the array contains
300,000 "foo"
string entries

Fire up the server and run the attack script.

```
$ node server.js &
$ node dos.js
```

You'll see that it can take V8 up to 10 seconds (varying depending on hardware) at times to parse such a large JSON string. This is bad, but thankfully it's exactly what the `limit()` middleware was designed to prevent.

BASIC USAGE

By adding the `limit()` middleware *before* the `bodyParser()` you can provide the size in bytes, or a human-readable string representation to specify the maximum size of the request body such as "1gb", "25mb", or "50kb". After running the server and attack script again, you'll see that Connect will terminate the request at 32 kilobytes.

```
var app = connect()
  .use(connect.limit('32kb'))
  .use(connect.bodyParser())
  .use(hello);
```

```
http.createServer(app).listen(3000);
```

WRAPPING LIMIT() FOR GREATER FLEXIBILITY

Limiting every request body to a small value like "32kb" is not feasible for applications accepting user uploads, as most image uploads will be larger than this, and files such as videos will definitely be much larger. However it may be a reasonable size for bodies formatted as JSON, XML, or similar. A good idea for applications needing to accept varying sizes of request bodies would be to wrap the `limit()` middleware based on some type of configuration. For example, you can wrap the middleware to specify a "Content-Type", which is shown in listing 8.2.

Listing 8.2 content-type-limiting.js: Wrap multiple limit() middleware based on the Content-Type of the request

```
function type(type, fn) {    1
  return function(req, res, next){
    var ct = req.headers['content-type'] || '';
    if (0 != ct.indexOf(type)) {    2
      return next();
    }
    fn(req, res, next);    3
  }
}

var app = connect()
  .use(type('application/x-www-form-urlencoded', connect.limit('64kb')))    4
  .use(type('application/json', connect.limit('32kb')))    4
  .use(type('image', connect.limit('2mb')))    5
  .use(type('video', connect.limit('300mb')))    6
  .use(connect.bodyParser())
  .use(hello);
```

- ① `fn` in this case is one of the `limit()` instances
- ② the returned middleware first checks the content-type
- ③ before invoking the passed-in `limit()` middleware
- ④ handles forms, json
- ⑤ image uploads up to 2 megabytes
- ⑥ video uploads up to 300 megabytes

Another way to use this middleware is to provide the `limit` option to

`bodyParser()` and the latter will call the `limit()` middleware transparently. The next middleware we are going to cover is a small, but very useful middleware which parses the requests' query-strings for your application to use.

8.1.4 `query: Query-string parser`

You have already learned about the `bodyParser` which can parse POST form requests, but what about the GET form requests? That's where the `query()` middleware comes in, which parses the query-string when present, and provides the `req.query` object for your application to use. For developers coming from PHP this is similar to the `$_GET` associative array. Much like `bodyParser()` it should be placed above any middleware that will use the property.

BASIC USAGE

Here you have an application utilizing the `query()` middleware that will respond with a JSON representation of the query-string sent by the request. Query-string parameters are usually used for controlling the display of the data being sent back.

```
var app = connect()
  .use(connect.query())
  .use(function(req, res, next){
    res.setHeader('Content-Type', 'application/json');
    res.end(JSON.stringify(req.query));
  });
}
```

Suppose you were designing a music library app. You could offer a search engine using the query-string to build up the search parameters, something like: `/songSearch?artist=Bob%20Marley&track=Jammin`. This example query would produce a `res.query` object like:

```
{ artist: 'Bob Marley', track: 'Jammin' }
```

This middleware uses the same third-party "qs" module as `bodyParser()`, so complex query-strings like `?images[]=foo.png&images[]=bar.png` produce the following object:

```
{ images: [ 'foo.png', 'bar.png' ] }
```

When no query-string parameters are given in the HTTP request, like

/songSearch, then `req.query` will default to an empty object:

```
{}
```

That's all there is to it! Next up you will learn about the built-in middleware that cover core web application needs like logging and sessions.

8.2 Middleware that implement core web application needs

Connect aims to implement and provide built-in middleware for all the most common web application needs, so that they don't need to be re-implemented over and over by each developer, which is error prone and tedious. "Core" web application concepts like logging, sessions, and virtual hosting are all provided by Connect out of the box.

In this section you will learn about 5 very useful middleware that will likely always be used in your applications:

- `logger` - a super flexible logging middleware to handle all your logging needs
- `favicon` - takes care of the `/favicon.ico` request without having to think about it
- `methodOverride` - transparent access to overwriting `req.method` for incapable clients
- `vhost` - set up multiple websites on a single server, a.k.a. "virtual hosting"
- `session` - session management middleware

Up until now you've created your own custom logging middleware, however it turns out that Connect provides a very flexible solution named `logger()`, so let's kick off exploring that first.

8.2.1 logger: Logging requests

Logger is a flexible request logging middleware with customizable log formats using "tokens", as well as options for buffering log output to decrease disk writes, and specifying a log stream for those who wish to log to a non-stdio stream such as a file or socket.

BASIC USAGE

To use the `logger()` middleware and use Connect's logging in your own application, simply invoke it as a function to return a logger middleware instance, which you can see in listing 8.3:

Listing 8.3 logger.js: Using the logger() middleware

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

var app = connect()
  .use(connect.logger())    ①
  .use(hello)               ②
  .listen(3000);
```

- ① With no arguments, the default logger options will be used
- ② 'hello' is assumed to be a middleware that responds with "Hello World"

By default the logger uses the following format, which is extremely verbose, however provides useful information about each HTTP request similar to how other web servers like Apache create their log files:

```
:remote-addr - - [:date] ":method :url HTTP/:http-version" :status
[→] :res[content-length] ":referrer" ":user-agent"
```

Each of the `:something` pieces are "tokens", which get replaced by real values from the HTTP request that is being logged. An example of this log format from a simple `curl(1)` request would output a line similar to the following:

```
127.0.0.1 - - [Wed, 28 Sep 2011 04:27:07 GMT] "GET / HTTP/1.1" 200 - "-"
[→] "curl/7.19.7 (universal-apple-darwin10.0)"
[→] libcurl/7.19.7 OpenSSL/0.9.8l zlib/1.2.3"
```

CUSTOMIZING LOG FORMATS

The most basic use of logger does not require any customization. However, you may want a custom format if you wanted to get other information, or be less verbose, or add custom output than what the default format provides. To do that you pass a custom string of tokens, where the format below would output something like "GET /users 15 ms".

```
var app = connect()
  .use(connect.logger(':method :url :response-time ms'))
  .use(hello);
```

By default the following tokens are available for use (note that the header names are not case-sensitive):

- :req[header] ex: :req[Accept]
- :res[header] ex: :res[Content-Length]
- :http-version
- :response-time
- :remote-addr
- :date
- :method
- :url
- :referrer
- :user-agent
- :status

Defining custom tokens is easy. All you have to do is provide a token name and callback function to the `connect.logger.token` function. For example, say you wanted to log each request's query-string. You might define it like this:

```
var url = require('url');
connect.logger.token('query-string', function(req, res){
  return url.parse(req.url).query;
});
```

The logger also comes with other predefined formats than the default one, like "short" and "tiny" for example. Another one of the predefined formats is "dev", which is designed specifically for development where it's often more useful to have concise output, since you are usually the only user on the site and you don't care about the details of the HTTP requests in that case. This format also color-codes the response status codes by type: responses with a status code in the 200s range will be green, 300s will be blue, 400s will be yellow, and 500s will be red. This color scheme makes it great for development. To use a predefined format you simply provide the name to `logger()`:

```
var app = connect()
  .use(connect.logger('dev'))
  .use(hello);
```

Now that you know how to format the logger, let's take a look at the optional

options object you may provide it.

LOGGER OPTIONS: 'STREAM', 'IMMEDIATE' AND 'BUFFER'

As mentioned previously `logger()` also provides some options to tweak how it behaves. One such option is `stream` allowing you to pass any node Stream instance that the logger will write to instead of `stdout`. You would want to do this if you wanted to direct the logger output to its own log file independent of your server's own output using an `fs.WriteStream`. When using options the "format" should be supplied via the object passed as well. The following example uses a custom format and logs to `"/var/log/myapp.log"` with the "append" flag so that the file is not truncated when the application boots.

```
var fs = require('fs')
var log = fs.createWriteStream('/var/log/myapp.log', { flags: 'a' })
var app = connect()
  .use(connect.logger({ format: ':method :url', stream: log }))
  .use('/error', error)
  .use(hello);
```

Another useful option is `immediate`, which will write the log line when the request is first received, rather than on the response. You might use this option if you are writing a server that keeps its requests open for a long time, and you want to know when the connection began, or for debugging a critical section of your app. This means that tokens such as `:status` and `:response-time` may not be used, as they are related to the response. To enable "immediate" mode just pass `true` for the `immediate` value, as shown here:

```
var app = connect()
  .use(connect.logger({ immediate: true }))
  .use('/error', error)
  .use(hello);
```

The third option available for is `buffer`, which is useful for when you want to minimize the number of writes to the disk where your log file is being written. This is especially useful if your log file is being written over a network, since you probably want to minimize the amount of network activity happening because of log files. The `buffer` option may be a number value which will be the interval in milliseconds to flush the buffer, or you may just pass `true` to use the default interval.

That's it for logging! Next up we're going to look at the favicon serving middleware.

8.2.2 favicon: Serving a favicon

A favicon is that tiny icon your browser displays in the address bar and bookmarks for your application. To get this icon, the browsers make a request for a file at `/favicon.ico`. These requests are often just a pain, so it's usually best to serve it as soon as possible, so the rest of your application can simply ignore them. The `favicon()` middleware will serve Connect's favicon by default (when no arguments are passed to it), which looks like the favicon outlined in figure 8.2:

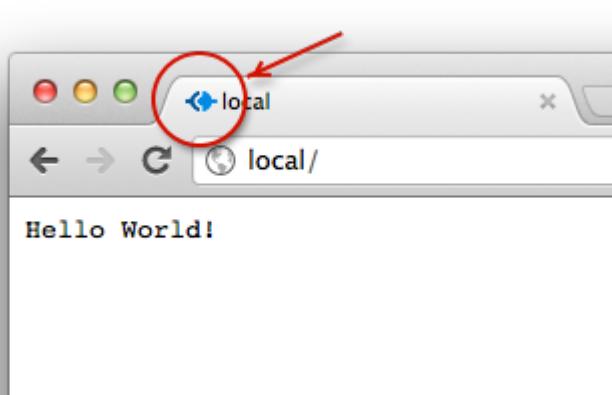


Figure 8.1 Connect's default "favicon"

BASIC USAGE

Typically this middleware is used at the very top of the stack, so even logging is ignored for favicon requests. The icon is then cached in memory for very fast subsequent requests. The following example shows `favicon()` using a manually specified custom `.ico` file by passing the file path as the only argument:

```
connect()
  .use(connect/favicon(__dirname + '/public/favicon.ico'))
  .use(connect/logger())
  .use(function(req, res) {
    res.end('Hello World!\n');
  });
}
```

Optionally, you may pass in a `maxAge` argument to specify how long browsers should cache the favicon in memory for. Next we have another small but helpful middleware `methodOverride()`, providing the means to "fake" the HTTP request method when client capabilities are limited.

8.2.3 `methodOverride`: Faking HTTP methods

An interesting problem arises in the browser when building a server that utilizes special HTTP verbs, like PUT or DELETE. The problem is that browser `<form>` methods may only be either GET or POST, restricting you from any other methods your application may be using. A common work-around for this behaviour is to add an `<input type=hidden>` with the value set to the method name you wish to use, and then have the server check that value and "pretend" it's the request method for this request. The `methodOverride()` middleware is the server-side half of this technique.

BASIC USAGE

By default this input name is "`_method`", however you may pass a custom value to `methodOverride()` as shown in the following snippet:

```
connect()
  .use(connect.methodOverride('__method__'))
  .listen(3000)
```

To demonstrate how `methodOverride()` is implemented, let's create a tiny application to update user information. The application will consist of a single form which will respond with a simple "success" message when the form is submitted by the browser and processed by the server, as illustrated in figure 8.3.:

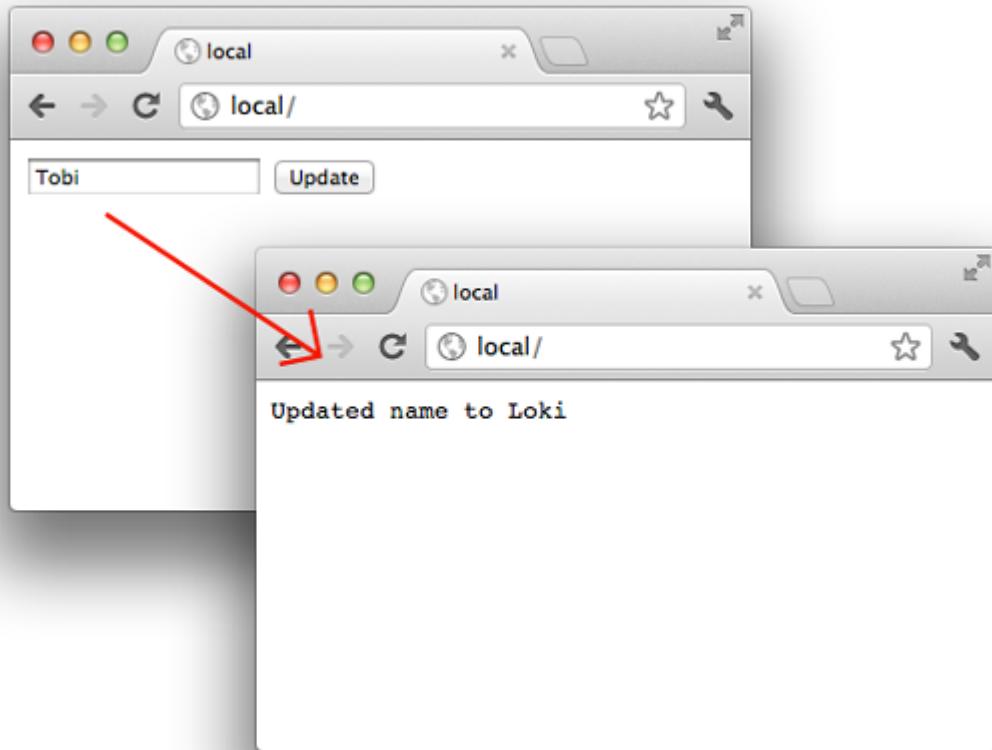


Figure 8.2 An example of using methodOverride to simulate a PUT request updating a form in the browser

The application handles updating the user data through the use of two separate middleware. In the update function next() is called when the request method is not PUT. As mentioned most user-agents will not respect the form attribute method= "put ", so this application in listing 8.4 will not function properly.

Listing 8.4 broken_user.js: Broken user update application

```

var connect = require('connect');

function edit(req, res, next) {
  if ('GET' != req.method) return next();
  res.setHeader('Content-Type', 'text/html');
  res.write('<form method="put">');
  res.write('<input type="text" name="user[name]" value="Tobi" />');
  res.write('<input type="submit" value="Update" />');
  res.write('</form>');
  res.end();
}

function update(req, res, next) {
  if ('PUT' != req.method) return next();
  res.end('Updated name to ' + req.body.user.name);
}

var app = connect()
  .use(connect.logger('dev'))
  .use(connect.bodyParser())
  .use(edit)
  .use(update);

app.listen(3000);

```

The update application would look something like listing 8.5. Here an additional input with the name "`_method`" was added to the form, and `methodOverride()` was added below the `bodyParser()` middleware as it references `req.body` to access the form data.

Listing 8.5 user_update.js: User update application with methodOverride implemented

```

var connect = require('connect');

function edit(req, res, next) {
  if ('GET' != req.method) return next();
  res.setHeader('Content-Type', 'text/html');
  res.write('<form method="post">');
  res.write('<input type="hidden" name="_method" value="put" />');
  res.write('<input type="text" name="user[name]" value="Tobi" />');
  res.write('<input type="submit" value="Update" />');
  res.write('</form>');
  res.end();
}

function update(req, res, next) {
  if ('PUT' != req.method) return next();
  res.end('Updated name to ' + req.body.user.name);
}

var app = connect()
  .use(connect.logger('dev'))
  .use(connect.bodyParser())
  .use(connect.methodOverride())
  .use(edit)
  .use(update)
  .listen(3000);

```

ACCESSING THE ORIGINAL REQ.METHOD

This middleware alters the original `req.method` property. However Connect copies over the original method which you may always access with `req.originalMethod`. So our form from above would output values like:

```

console.log(req.method);
// "PUT"
console.log(req.originalMethod);
// "POST"

```

This may seem like quite a bit of work for a simple form, but we promise this will be more enjoyable when we discuss higher-level features from Express in

Chapter 9 and templating in Chapter 10. The next middleware we're going to look at is `vhost()`, which is a small middleware for serving applications based on hostnames.

8.2.4 `vhost: Virtual hosting`

The `vhost()` or "virtual host" middleware provides a very simple light-weight way to route via the "Host" request header. This task is commonly performed by a reverse proxy which then forwards the request to a web server running locally on a different port. However the `vhost()` middleware does this in the same process by passing control to the node HTTP server passed to the `vhost` instance.

BASIC USAGE

Like all the middleware that Connect provides out of the box, a single line is all it takes to get up and running with the `vhost()` middleware. It takes 2 arguments. The first is a string that is the hostname that this `vhost` instance will match against. The second argument is the actual `http.Server` (a Connect "app" is a subclass of `http.Server`, so those work as well) instance to use when an HTTP request with a matching hostname occurs.

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

var server = connect()
var app = require('./sites/expressjs.dev');

server.use(connect.vhost('expressjs.dev', app));

server.listen(3000);
```

In order to require the HTTP server instance, the `module.exports` of `./sites/expressjs.dev` must look something like the following:

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

module.exports = http.createServer(function(req, res){
  res.end('hello from expressjs.com\n');
});
```

USING MULTIPLE VHOST() INSTANCES

Like any other middleware, you can use `vhost()` more than once in order to map several hosts to their associated applications:

```
var app = require('./sites/expressjs.dev');
server.use(connect.vhost('expressjs.dev', app));

var app = require('./sites/learnboost.dev');
server.use(connect.vhost('learnboost.dev', app));
```

Rather than setting up the `vhost()` middleware manually like this, you could also generate a list of the hosts from the filesystem as shown here with the `fs.readdirSync()` method that returns an array of directory entries.

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

var app = connect()
var sites = fs.readdirSync('source/sites');

sites.forEach(function(site){
  console.log(' ... %s', site);
  app.use(connect.vhost(site, require('./sites/' + site)));
});

app.listen(3000);
```

Simplicity is the benefit of using `vhost()` instead of a reverse proxy, allowing you to manage all of your applications as a single unit. This is ideal for serving several smaller sites, or sites that are largely comprised of static content, but also has the downside that if one site causes a crash then all of your sites will be taken down (since they all run in the same process).

Next we're going to take a look at the one of the most fundamental middleware that Connect provides, the session management middleware appropriately named `session()`, which relies on `cookieParser()` for cookie signing.

8.2.5 session: Session management

In chapter 4 "Building Node Web Applications" we discussed how Node provides all the means to build concepts like "sessions", however it does not provide them out of the box. Following Node's general philosophy of a small core, and large user-land, session management can be created as a third-party addon to Node. And that's exactly what this middleware is for.

Connect's `session()` middleware provides robust, intuitive, and community-backed session management with numerous session stores ranging from the default memory store, redis, mongodb, couchdb, and cookie-based stores.

In this section we'll look at setting up the middleware, how to work with session data, and utilizing the Redis key/value store as an alternate session store.

First let's set up the middleware and explore the options available.

BASIC USAGE

As previously mentioned, the `session()` middleware requires signed cookies to function, so you should use `cookieParser()` somewhere above it and pass a secret.

Listing 8.6 implements a small page view count application minimal setup where no options are passed to `session()` at all, and the default in-memory data-store is used. By default the cookie name is "connect.sid", and is set to be "httpOnly" meaning client-side scripts cannot access its value. However these are options you may tweak as you'll see next.

Listing 8.6 page_view_counter.js: Connect page view counter using sessions

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

var app = connect()
  .use(connect.favicon())
  .use(connect.cookieParser('keyboard cat'))
  .use(connect.session())
  .use(function(req, res, next){
    var sess = req.session;
    if (sess.views) {
      res.setHeader('Content-Type', 'text/html');
      res.write('<p>views: ' + sess.views + '</p>');
      res.end();
      sess.views++;
    } else {
      sess.views = 1;
      res.end('welcome to the session demo. refresh!');
    }
  });
app.listen(3000);
```

SETTING THE SESSION EXPIRATION DATE

Suppose you want sessions to expire in 24 hours, to send the session cookie only when HTTPS is used, and to configure the cookie name. You might pass an object like the one shown here:

```

var hour = 3600000;
var sessionOpts = {
  key: 'myapp_sid',
  cookie: { maxAge: hour * 24, secure: true }
};

...
  .use(connect.cookieParser('keyboard_cat'))
  .use(connect.session(sessionOpts))
...

```

When using Connect (and, as you'll see in the next chapter, Express) you'll often set `maxAge`, accepting the number of milliseconds from that given point in time. This method of expressing future dates is often more intuitive, essentially expanding to `new Date(Date.now() + maxAge)`.

Now that sessions are set up, let's look at all of the methods and properties available when working with session data.

WORKING WITH SESSION DATA

Connect's session data management is very simple. The basic principle is: any properties assigned to the `req.session` object are saved when the request is complete, then loaded on subsequent requests from the same user (browser). For example saving shopping-cart information is as simple as assigning an object to the `cart` property as shown here.

```
req.session.cart = { items: [1,2,3] };
```

The next time you access `req.session.cart` on subsequent requests you'll have the `.items` array available. Because this is just a regular JavaScript object you can call methods on the nested objects on subsequent requests as shown here, and they'll be saved as you expect.

```
req.session.cart.items.push(4);
```

One important thing to keep in mind is that this session object gets serialized as JSON in between requests, therefore the `req.session` object follows the same restrictions as JSON such as cyclic properties not being allowed, function objects may not be used, and `Date` objects not being serialized correctly. So be sure to be mindful of those restrictions when using the session object.

While Connect will save session data for you automatically, internally it's calling the `Session#save([callback])` method, which is available as public API as well. Two additional helpful methods are the `Session#destroy()` and `Session#regenerate()` methods, which are often used when authenticating a user to prevent session fixation³.

Footnote 3 https://www.owasp.org/index.php/Session_fixation

When you build some applications with Express in later chapters you'll be using these methods for authentication. But for now, let's move on to manipulating session cookies.

MANIPULATING SESSION COOKIES

Connect allows you to provide global cookie settings for sessions, but it's also possible to manipulate a specific cookie via the `Session#cookie` object, which defaults to the global settings.

Before you start tweaking properties, let's extend the previous session application to inspect the session cookie properties by writing each property into individual `<p>` tags in the response HTML as shown here:

```
...
res.write('<p>views: ' + sess.views + '</p>');
res.write('<p>expires in: ' + (sess.cookie.maxAge / 1000) + 's</p>');
res.write('<p>httpOnly: ' + sess.cookie.httpOnly + '</p>');
res.write('<p>path: ' + sess.cookie.path + '</p>');
res.write('<p>domain: ' + sess.cookie.domain + '</p>');
res.write('<p>secure: ' + sess.cookie.secure + '</p>');
...
...
```

Connect allows all of the cookie properties such as "expires", "httpOnly", "secure", "path", "domain" to be altered programmatically per-session. For example expiring an active session in 5 seconds would look like this:

```
req.session.cookie.expires = new Date(Date.now() + 5000);
```

An alternative, more intuitive API for expiry is the `.maxAge` accessor, which allows you to get and set the value in milliseconds relative to the current time. The following will also expire the session in 5 seconds:

```
req.session.cookie.maxAge = 5000;
```

The remaining properties "domain", "path", and "secure" limit the cookie "scope", restricting it by domain, path, or to secure connections; while "httpOnly" prevents client-side scripts from accessing the cookie data. These properties can be manipulated in the same manner:

```
req.session.cookie.path = '/admin';
req.session.cookie.httpOnly = false;
```

So far you've been using the default memory store to store session data, so let's take a look at how you can plugin-in alternative data-stores.

SESSION STORES

By default the built-in `connect.session.MemoryStore` is used, a simple in-memory data store which is ideal for running application tests as no other dependencies are necessary. During development, and of course in production it's best to have a persistent, scalable database backing your session data.

While essentially any database can act as a session store, typically low-latency key/value stores work best for such volatile data. The Connect community has created several session stores for databases including CouchDB, MongoDB, Redis, Memcached, PostgreSQL and others.

Here you'll be using Redis with the "connect-redis" module. You learned about Redis in-depth and interacting with it using the "node_redis" module in Chapter 5. Now you will learn how to use Redis to store your session data in Connect. Redis is a good backing store as it supports key expiration, great performance, and it's easy to install. You should have Redis installed and running from chapter 5, but try invoking the `redis-server` command just to be sure.

```
$ redis-server
[11790] 16 Oct 16:11:54 * Server started, Redis version 2.0.4
[11790] 16 Oct 16:11:54 * DB loaded from disk: 0 seconds
[11790] 16 Oct 16:11:54 * The server is now ready to accept
→ connections on port 6379
[11790] 16 Oct 16:11:55 - DB 0: 522 keys (0 volatile) in 1536 slots HT.
```

Next you'll want to install "connect-redis" by adding it to your `package.json` and running `npm install`, or execute `npm install connect-redis` directly. The "connect-redis" module exports a function which should be passed `connect` as shown here:

```

var connect = require('connect')
var RedisStore = require('connect-redis')(connect);

var app = connect()
  .use(connect.favicon())
  .use(connect.cookieParser('keyboard cat'))
  .use(connect.session({ store: new RedisStore({ prefix: 'sid' }) }))
...

```

Passing in the `connect` reference to `connect-redis` allows it to inherit from `connect.session.Store.prototype`. This is important because in Node, a single process may use multiple different versions of a module at once, so by passing your specific version of Connect you can be sure that `connect-redis` uses the proper copy. The instance of `RedisStore` is passed to `session()` as the `store` value, and any options desired such as a key prefix for your sessions may be passed to the `RedisStore` constructor.

Whew! Well `session` was a lot to cover, and that finishes up all the "core" concept middleware. Next up we'll go over the built-in middleware that handle web application security, a very important subject for applications needing to secure their data.

8.3 Middleware that handle web application security

As we have stated many times, Node's core API is all about being low-level and unopinionated. Unfortunately this means that it provides no built-in security or best practices when it comes to building web application. However that's where Connect steps in to implement these security practices for use in your Connect applications.

This section will teach you about 3 more of Connect's built-in middleware, this time with a focus on security:

- `basicAuth` - HTTP Basic authentication middleware for protecting data
- `csrf` - middleware implementing "Cross-site request forgery" protection
- `errorHandler` - helps you debug during development

The first security middleware we will talk about is `basicAuth()`, implementing HTTP Basic authentication for restricted areas of your application.

8.3.1 basicAuth: HTTP Basic authentication

In Chapter 7's "Mounting middleware & servers" section you created a crude implementation of a basic authentication middleware. Well it turns out that Connect provides a real implementation of this out of the box! As previously mentioned basic auth is a very simple HTTP authentication mechanism, which should be used with caution as it can be trivial for an attacker to intercept unless served over HTTPS. That being said it can be useful for adding "quick and dirty" authentication to a small or personal application.

When your application has the `basicAuth()` middleware in use, web browsers will prompt for credentials the first time the user attempts to connect to your application as shown in figure 8.4:

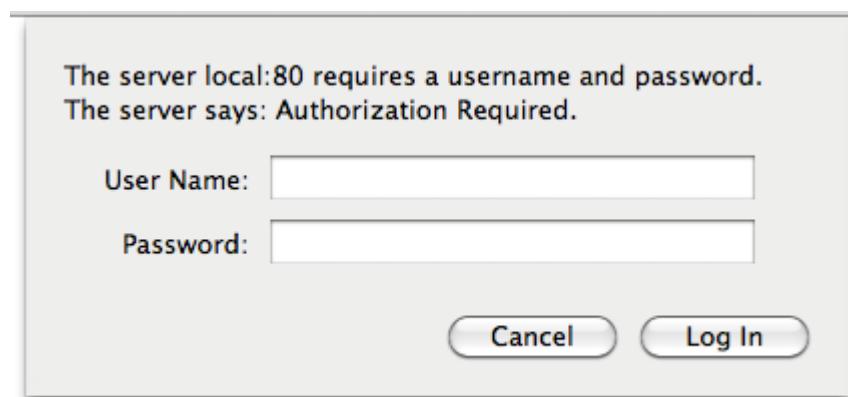


Figure 8.3 Basic authentication prompt

BASIC USAGE

This middleware provides three means of validating the credentials given. The first is to provide a single username and password as shown here:

```
var app = connect()
  .use(connect.basicAuth('tj', 'tobi'));
```

PROVIDING A CALLBACK FUNCTION

The second is to provide a callback, which must return `true` in order to succeed. This is useful for checking the credentials against a hash or similar.

```
var users = {
  tobi: 'foo',
  loki: 'bar',
  jane: 'baz'
```

```

};

var app = connect()
  .use(connect.basicAuth(function(user, pass){
    return users[user] === pass;
}));
```

PROVIDING AN ASYNCHRONOUS CALLBACK FUNCTION

The final option is similar, except this time a callback is passed with three arguments defined which enables the use of asynchronous lookups, useful when authenticating from a file on disk, or when querying from a database as shown in listing 8.7.

Listing 8.7 basicAuth_async.js: Connect basicAuth middleware doing asynchronous lookups

```

var app = connect();

app.use(connect.basicAuth(function(user, pass, callback){
  User.authenticate({ user: user, pass: pass }, gotUser); ①

  function gotUser(err, user) { ②
    if (err) return callback(err);
    callback(null, user); ③
  }
}));
```

- ① User.authenticate could be a database validation function
- ② Invoked asynchronously when the database has responded
- ③ Provide the basicAuth callback with the 'user' object from the database

AN EXAMPLE WITH CURL(1)

Suppose you want to restrict access to all requests coming to your server. You might set up the application like this:

```

var connect = require('connect');

var app = connect()
  .use(connect.basicAuth('tobi', 'ferret'))
  .use(function (req, res) {
    res.end("I'm a secret\n");
});
```

```
app.listen(3000);
```

Now try issuing an HTTP request to the server with `curl(1)` and you'll see that you are unauthorized:

```
$ curl http://localhost -i
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="Authorization Required"
Connection: keep-alive
Transfer-Encoding: chunked

Unauthorized
```

Now issuing the same request with HTTP Basic authorization credentials (notice the beginning of the URL) will provide access:

```
$ curl --user tobi:ferret http://localhost -i
HTTP/1.1 200 OK
Date: Sun, 16 Oct 2011 22:42:06 GMT
Cache-Control: public, max-age=0
Last-Modified: Sun, 16 Oct 2011 22:41:02 GMT
ETag: "13-1318804862000"
Content-Type: text/plain; charset=UTF-8
Accept-Ranges: bytes
Content-Length: 13
Connection: keep-alive

I'm a secret
```

Continuing on with the security theme of this section, let's look at the middleware named `csrf()` to help protect against Cross-site request forgery.

8.3.2 csrf: Cross-site request forgery protection

Cross-site request forgery (or CSRF for short) is a form of attack where an un-trusted site exploits the trust that a site has with the web browser. The attack works by having an authenticated user on your application visit a different site that an attacker has either created or compromised, and then making requests on the users behalf without them knowing about it. This is commonly known as "cross site scripting", or CSRF for short.

This is a complicated attack, so let's try to explain it with an example. Suppose a user is logged into your application, and that in your application the request `DELETE /account` would trigger your account to be destroyed (though only

while you're logged in). Now suppose that user visits a forum that happens to be vulnerable to XSS. A malicious user could post a script that issues the `DELETE /account` request, thus destroying your account. This is a bad situation for your application to be in, and thankfully the `csrf()` middleware can help you protect against such an attack.

The `csrf()` middleware works by generating a 24 character unique id, the "authenticity token", assigning it to the user's session as `req.session._csrf`. This token can then be placed as a hidden form input named "`_csrf`" so that the CSRF middleware can validate the token on submission, repeating this behaviour per interaction.

BASIC USAGE

To ensure that `csrf()` may access `req.body._csrf`, the hidden input value, and `req.session._csrf`, you'll want to make sure that you add the middleware below `bodyParser()` and `session()` as shown in the following example:

```
connect()
  .use(connect.bodyParser())
  .use(connect.cookieParser('secret'))
  .use(connect.session())
  .use(connect.csrf());
```

Another aspect of web development is having verbose logs and detailed error reporting available both in production and development environments. Let's take a look at the `errorHandler()` middleware which is designed for exactly that.

8.3.3 `errorHandler`: Development error handling

The `errorHandler()` middleware bundled with Connect is ideal for development, providing verbose HTML, JSON, and plain-text error response based on the "Accept" header field. It's really only meant for use during development and should not be part of the production configuration.

BASIC USAGE

Typically this middleware should be the last used so it may "catch" all errors as shown here:

```
var app = connect()
  .use(connect.logger('dev'))
  .use(function(req, res, next){
```

```

    setTimeout(function () {
      next(new Error('something broke!'));
    }, 500);
})
.use(connect.errorHandler());

```

RECEIVING AN HTML ERROR RESPONSE

If you were to view any page in your browser with the setup shown here you'll see a Connect error page like shown in figure 8.5, displaying the error message, response status, and the entire stack trace.

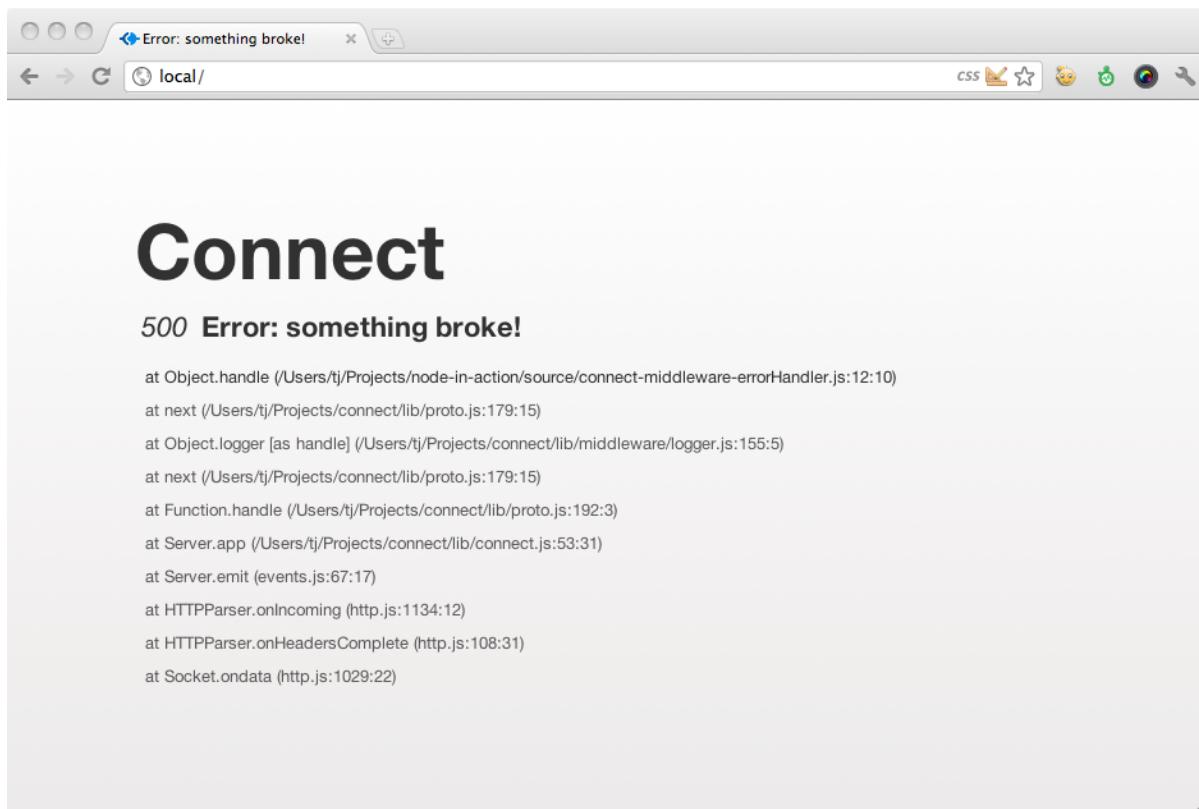


Figure 8.4 Connect's default `errorHandler()` middleware as displayed in a web browser

RECEIVING A PLAIN-TEXT ERROR RESPONSE

Now suppose you're testing an API built with Connect, it's far from ideal to respond with a large chunk of HTML, so by default `errorHandler()` will respond with "text/plain", which is ideal for command-line HTTP clients such as `curl(1)` as illustrated in the following stdout:

```

$ curl http://localhost/
Error: something broke!
  at Object.handle (/Users/tj/Projects/node-in-action/source
   /connect-middleware-errorHandler.js:12:10)

```

```

at next (/Users/tj/Projects/connect/lib/proto.js:179:15)
at Object.logger [as handle] (/Users/tj/Projects/connect
  ↴ /lib/middleware/logger.js:155:5)
at next (/Users/tj/Projects/connect/lib/proto.js:179:15)
at Function.handle (/Users/tj/Projects/connect/lib/proto.js:192:3)
at Server.app (/Users/tj/Projects/connect/lib/connect.js:53:31)
at Server.emit (events.js:67:17)
at HTTPParser.onIncoming (http.js:1134:12)
at HTTPParser.onHeadersComplete (http.js:108:31)
at Socket.ondata (http.js:1029:22)

```

RECEIVING A JSON ERROR RESPONSE

Now if you send an HTTP request that has the `Accept: application/json` HTTP header, you'll get the following JSON response:

```

$ curl http://localhost/ -H "Accept: application/json"
{"error": {"stack": "Error: something broke!\n
  ↴     at Object.handle (/Users/tj/Projects/node-in-action
  ↴ /source/connect-middleware-errorHandler.js:12:10)\n
  ↴     at next (/Users/tj/Projects/connect/lib/proto.js:179:15)\n
  ↴     at Object.logger [as handle] (/Users/tj/Projects
  ↴ /connect/lib/middleware/logger.js:155:5)\n
  ↴     at next (/Users/tj/Projects/connect/lib/proto.js:179:15)\n
  ↴     at Function.handle (/Users/tj/Projects/connect/lib/proto.js:192:3)\n
  ↴     at Server.app (/Users/tj/Projects/connect/lib/connect.js:53:31)\n
  ↴     at Server.emit (events.js:67:17)\n
  ↴     at HTTPParser.onIncoming (http.js:1134:12)\n
  ↴     at HTTPParser.onHeadersComplete (http.js:108:31)\n
  ↴     at Socket.ondata (http.js:1029:22)", "message": "something broke!"}}

```

We have added additional formatting to the JSON response, for the sake of being able to read it easier, but when Connect actually sends the JSON response it gets compacted nicely by getting run through `JSON.stringify()`.

Are you feeling like a Connect security guru now? Maybe not yet, but you should have enough of the basics down to make your applications secure, all using Connect's built-in middleware. Next up, let's move on to a very common web app need: serving static files.

8.4 Middleware for serving static files

In Chapter 4 we talked about sessions being one example of a higher-level concept that was not provided in Node core. Well serving static files is another one of those requirements common to many web applications, but also not provided by Node Core. Connect has you covered here as well!

Coming up in this next section you will learn about 3 more of Connect's built-in

middleware focusing on serving files from the filesystem, much like regular HTTP servers are intended:

- `static` - serves files from the filesystem from a given root directory
- `compress` - compresses responses ideal for use with `static`
- `directory` - serves pretty directory listings when a directory is requested

First up, we'll show you how to serve static files with a single line of code using the `static` middleware.

8.4.1 static: Static file serving

Connect's `static()` middleware implements a high performance, flexible, feature-rich static file server supporting HTTP cache mechanisms, Range requests and more. Even more important are the security checks for malicious paths, disallowing access to hidden files (beginning with a `.`) by default, and rejecting poison null bytes⁴. In essence, `static()` is a very secure and compliant static file serving middleware, ensuring the best compatibility with the various HTTP clients out there in the world.

Footnote 4 <http://insecure.org/news/P55-07.txt>

BASIC USAGE

Suppose your application follows the typical scenario of serving static assets from a directory named `"/public"`. This can be achieved with a single line of code:

```
app.use(connect.static('public'));
```

With this configuration `static()` will check if a regular file of that name exists in `"/public/"`. If so, the response Content-Type field value will be defaulted based on the file's extension, and the data will be transferred. If the requested path does not represent a file, the `next()` callback will be invoked, allowing subsequent middleware (if any) to handle the request.

To test it out create a file named `"/public/foo.js"` with `console.log('tobi')`, and issue a request to the server using `curl(1)` with the `-i` flag, telling it to print the HTTP headers. You'll see that HTTP cache related header fields are set appropriately, the Content-Type reflects the `".js"` extension, and the content is transferred.

```
$ curl http://localhost/foo.js -i
HTTP/1.1 200 OK
Date: Thu, 06 Oct 2011 03:06:33 GMT
Cache-Control: public, max-age=0
Last-Modified: Thu, 06 Oct 2011 03:05:51 GMT
ETag: "21-1317870351000"
Content-Type: application/javascript
Accept-Ranges: bytes
Content-Length: 21
Connection: keep-alive

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

Since the request path is used as-is, files nested within directories are served as you would expect. For example you may get a "GET /javascripts/jquery.js" request and a "GET /stylesheets/app.css" request for on your server, which would server the files "./public/javascripts/jquery.js" and "./public/stylesheets/app.css" respectively.

USING STATIC() WITH "MOUNTING"

Sometimes applications prefix pathnames with "/public", "/assets", "/static" or similar. With the mounting concept that Connect implements serving static files from multiple directories is simple. Just mount the app at the location you want. As mentioned in Chapter 7, the middleware itself has no knowledge that it is mounted, as the prefix is removed. For example, a request to GET "/app/files/js/jquery.js" with `static()` mounted at "/app/files" will simply appear to the middleware as "GET /js/jquery". This works out great for the prefixing functionality because "/app/files" will not be part of the file resolution.

```
app.use('/app/files', connect.static('public'));
```

What you'll see now is that the original request of "GET /foo.js" will no longer work, as the middleware is not invoked unless the mount-point is present, but the prefixed version "GET /app/files/foo.js" will transfer the file.

```
$ curl http://localhost/foo.js
Cannot get /foo.js

$ curl http://localhost/app/files/foo.js
console.log('tobi');
```

ABSOLUTE VS. RELATIVE DIRECTORY PATHS

Keep in mind that the path passed into the `static()` middleware is relative to the current working directory. So passing in "public" as your path will essentially resolve to `process.cwd() + "public"`. Sometimes though you may want to use absolute paths when specifying the base directory, and the `__dirname` variable helps with that:

```
app.use('/app/files', connect.static(__dirname + '/public'));
```

SERVING "INDEX.HTML" WHEN A DIRECTORY IS REQUESTED

Another useful feature provided by `static()` is its ability to serve `index.html` files. When a request for a directory is made and an `index.html` file lives in that directory, it will be served.

Now that you can serve static files with a single line of code, let's take a look at how you can compress the response data using the `compress()` middleware to decrease the amount of data being transferred.

8.4.2 `compress`: Compressing static files

The `zlib` module, which provides developers with mechanisms for compressing and decompressing data with `gzip` and `deflate`. Connect 2.0 and above provide this at the HTTP server level for compressing outgoing data with the `compress()` middleware.

The `compress()` middleware auto-detects accepted encodings via the "Accept-Encoding" header field. If this field is not present then the identity encoding is used, meaning the response is untouched. Otherwise if the field contains "gzip", "deflate", or both then the response will be compressed.

BASIC USAGE

You should generally add this middleware high in the Connect stack, as it wraps the `res.write()` and `res.end()` methods. Here the static files served will support compression:

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

var app = connect()
  .use(connect.compress())
  .use(connect.static('source'));
```

```
app.listen(3000);
```

In the snippet that follows, a small 189 byte JavaScript file is served. By default curl(1) does not send the "Accept-Encoding" field so we receive plain-text:

```
$ curl http://localhost/script.js -i
HTTP/1.1 200 OK
Date: Sun, 16 Oct 2011 18:30:00 GMT
Cache-Control: public, max-age=0
Last-Modified: Sun, 16 Oct 2011 18:29:55 GMT
ETag: "189-1318789795000"
Content-Type: application/javascript
Accept-Ranges: bytes
Content-Length: 189
Connection: keep-alive

console.log('tobi');
console.log('loki');
console.log('jane');
console.log('tobi');
console.log('loki');
console.log('jane');
console.log('tobi');
console.log('loki');
console.log('jane');
```

Now the following curl(1) command adds the "Accept-Encoding" field showing that it's willing to accept gzip-compressed data. As you can see even for such a small file the data transferred is reduced considerably as the data is quite repetitive.

```
$ curl http://localhost/script.js -i -H "Accept-Encoding: gzip"
HTTP/1.1 200 OK
Date: Sun, 16 Oct 2011 18:31:45 GMT
Cache-Control: public, max-age=0
Last-Modified: Sun, 16 Oct 2011 18:29:55 GMT
ETag: "189-1318789795000"
Content-Type: application/javascript
Accept-Ranges: bytes
Content-Encoding: gzip
Vary: Accept-Encoding
Connection: keep-alive
Transfer-Encoding: chunked

K??+??I???O?P/?O?T?JF?????J?K???v? !_?
```

You could try the same with "Accept-Encoding: deflate" as well. Gzip is

DEFLATE wrapped with a header, the DEFLATE payload and a checksum used for data integrity.

USING A CUSTOM FILTER FUNCTION

By default `compress()` supports the mime types "text/*", "*json", and "*javascript", as defined in the default `filter` function:

```
exports.filter = function(req, res){
  var type = res.getHeader('Content-Type') || '';
  return type.match(/json|text|javascript/);
};
```

To alter this behaviour you can pass a `filter` in the options object, as shown in the following snippet where only plain text will be compressed:

```
function filter(req) {
  var type = req.getHeader('Content-Type') || '';
  return 0 == type.indexOf('text/plain');
}

connect()
  .use(connect.compress({ filter: filter }))
```

SPECIFYING THE COMPRESSION AND MEMORY LEVELS

Node's `zlib` bindings also provide options for tweaking performance and compression characteristics which may also be passed to the `compress()` function. Here the `compression level` is set to 3 for less but faster compression, and `memLevel` is set to 8 for faster compression through using more memory. These values entirely depend on your application and the resources available to it. You may consult Node's `zlib` documentation for details.

```
connect()
  .use(connect.compress({ level: 3, memLevel: 8 }))
```

Next up is the `directory()`, a small middleware complementing `static()` to serve directory listings in all kinds of formats.

8.4.3 `directory`: Directory listings

Connect's `directory()` middleware is a small directory listing middleware, providing a way for users to browse remote files. Figure 8.6 illustrates the interface provided by this middleware, complete with a file search input, file icons, and clickable bread-crumbs.

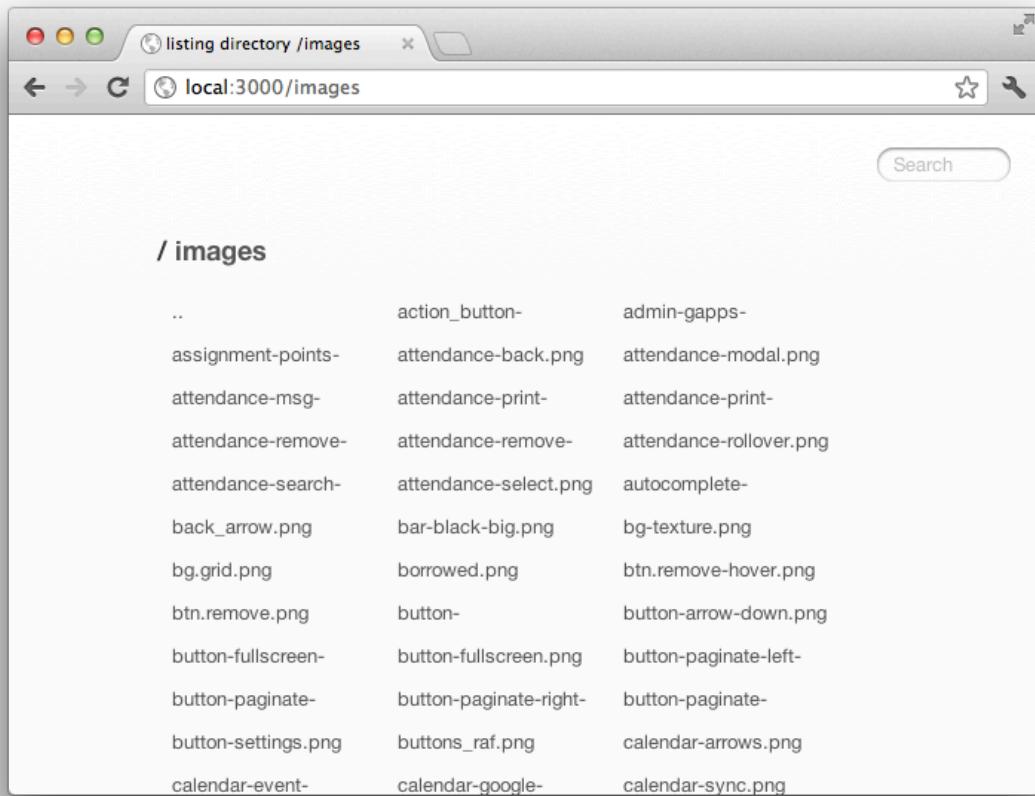


Figure 8.5 Serve directory listings with Connect's `directory()` middleware

BASIC USAGE

This middleware is designed to work with the `static()` middleware, which will perform the actual file serving, while `directory()` simply serves the listings. A setup can be as simple as the following snippet, where the `./public` directory is served. This means the request "GET /" will serve this directory.

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

var app = connect()
  .use(connect.directory('public'))
```

```
.use(connect.static('public'));

app.listen(3000);
```

USING DIRECTORY() WITH "MOUNTING"

Through the use of middleware mounting you can prefix both the `directory()` and `static()` middleware to any path you like, in this case "GET /files". Here the `icons` option is used to enable icons, and `hidden` is enabled for both middleware to allow viewing and serving of hidden files.

```
var app = connect()
  .use('/files', connect.directory('public',
    { icons: true, hidden: true }))
  .use('/files', connect.static('public', { hidden: true }));

app.listen(3000);
```

It is now be possible to navigate through files and directories with ease!

8.5 Summary

The real power of Connect comes from its rich suite of bundled reusable middleware that provide implementations for common web application concepts like session management, robust static file serving, and compressing outgoing data, among others. The goal there is to give the developers some batteries included out of the box, so that everyone isn't constantly rewriting the same pieces of code (possibly less efficiently) for their own application or framework.

Connect is perfectly capable of building entire applications using combinations of middleware, as you have seen throughout this chapter. However, Connect is intentionally unopinionated, meaning that it doesn't bundle any middleware that might be considered "high level", like routing. This low-level approach makes Connect great as a base for higher level frameworks, which is exactly how Express integrates with it.

You might be thinking, why not just use Connect for building a web application? While that's perfectly possible, the higher-level Express web frameworks makes full use of Connect's functionality, but takes application development one step further. Express will make application development quicker and more enjoyable with an elegant view system, powerful routing, and several request and response related methods. Let's explore Express in the next chapter.

A large, light gray graphic element consisting of a stylized number '9' and the word 'Express' written in a lowercase, sans-serif font.

This chapter covers

- Starting a new Express application
- Configuring your application
- Creating Express views
- Handling file uploads and downloads

Things are about to get even more fun. The Express¹ web framework is built on top of Connect, providing tools and structure that make writing web applications easier, faster, and more fun. This includes features such as a unified view system that lets you use nearly any template engine you wish, simple utilities for responding to various data formats, content negotiation, file transfer, routing, and more.

Footnote 1 <http://expressjs.com>

In comparison to application frameworks such as Django or Ruby on Rails, Express is extremely small. The philosophy behind Express is that applications vary greatly in requirements and implementations, so with a lightweight framework you can craft exactly what you need, and nothing more. This philosophy is true within Express, as well as within the entire Node community, which is focused on smaller, more modular bits of functionality rather than monolithic frameworks.

Throughout this chapter, you're going to learn how to use Express to create applications by building a stock photo-sharing application from start to finish. During the build you'll learn how to:

- Generate the initial application structure
- Configure Express and your application
- Render views and template engine integration
- Handle forms and file uploads

The final stock photo application will have a list view that will look like figure 9.1:

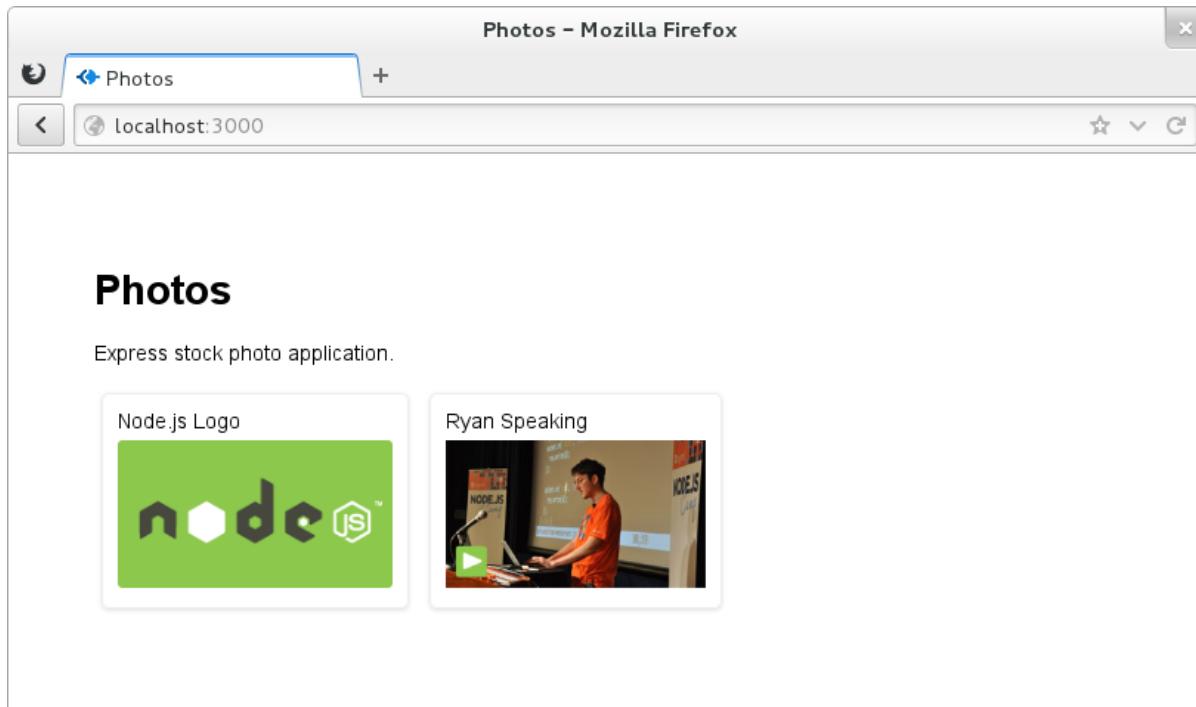


Figure 9.1 The photo list view

It'll include a form for uploading new photos, as shown in figure 9.2:

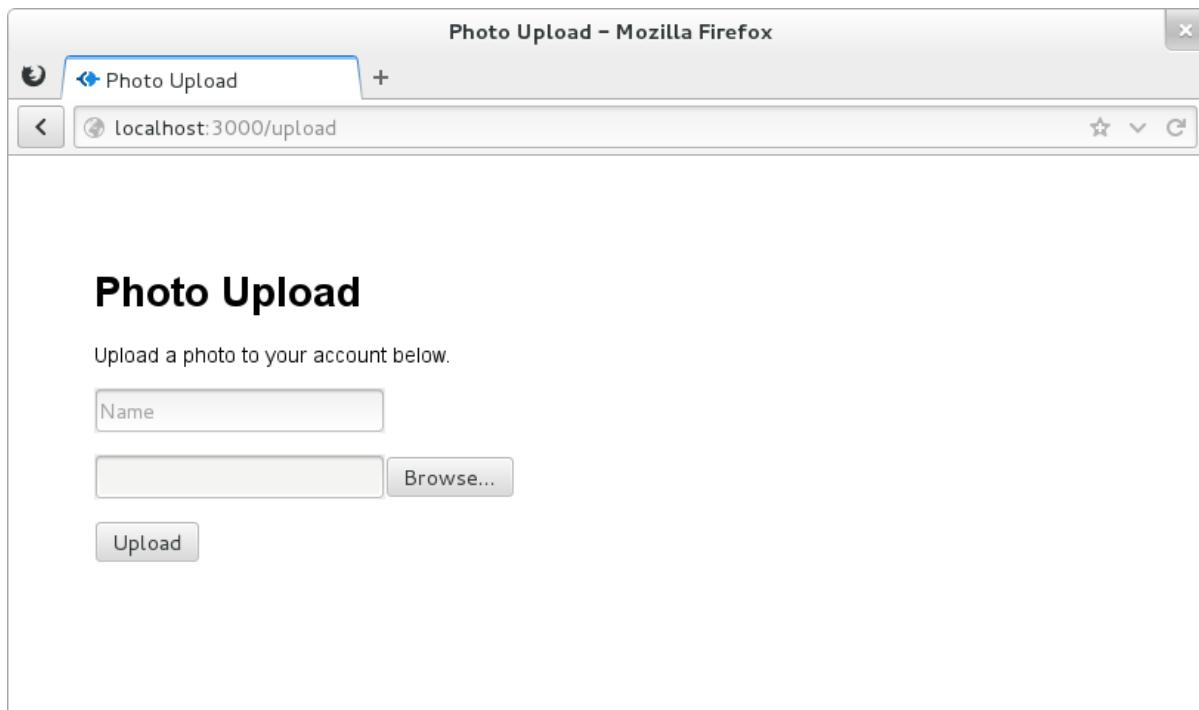


Figure 9.2 The photo upload view

It'll also include a mechanism for downloading photos, as shown in figure 9.3:

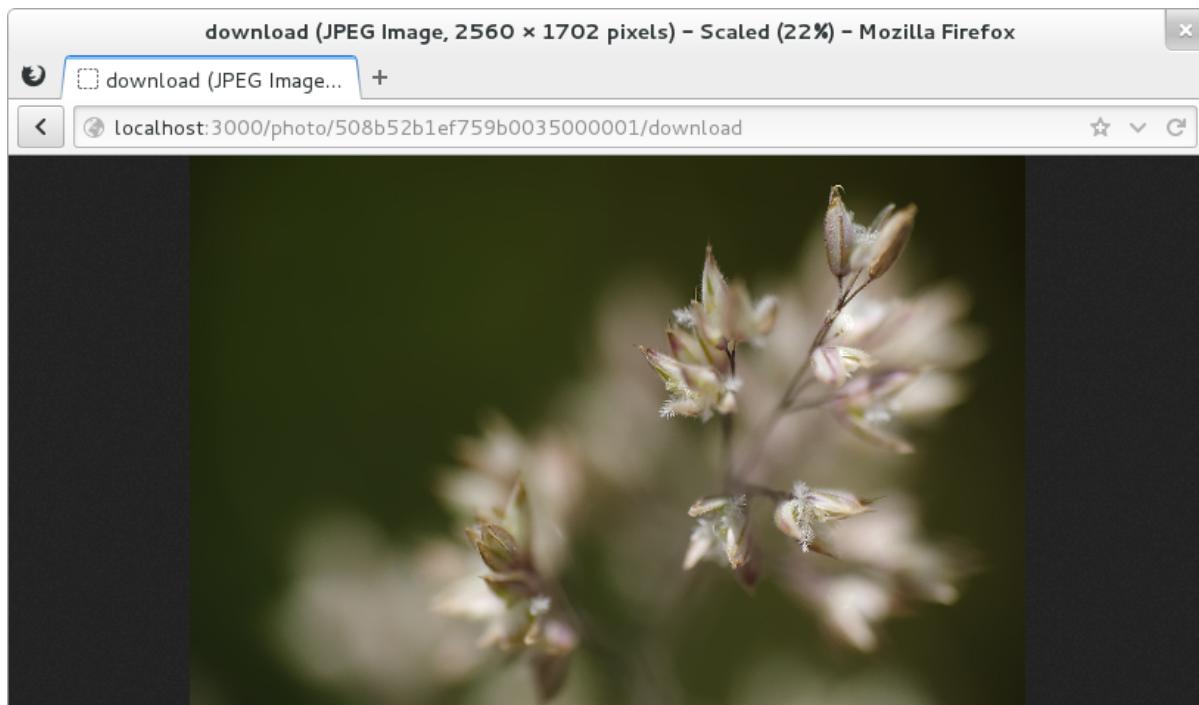


Figure 9.3 Downloading a file

9.1 Generating the application skeleton

Express doesn't force application structure on the developer—you can place routes in as many files as you wish, public assets in any directory you wish, and so on. A minimal Express application may be as little as listing 9.1, which implements a fully functional HTTP server.

Listing 9.1 A minimal Express application

```
var express = require('express');
var app = express();
app.get('/', function(req, res){    ①
    res.send('Hello');                ②
});
app.listen(3000);                   ③
```

- ① Respond to any web request to “/”
- ② Send “Hello” as response text
- ③ Listen on port 3000

The `express(1)` executable script bundled with Express can set up an application skeleton for you. The generated application is a good way to get started if you're new to Express, as it sets up an application complete with templates, public assets, configuration, and more.

The default application skeleton that `express(1)` generates consists of only a few directories and files, as shown in figure 9.4. This structure is designed to get developers up and running with Express in seconds, but the application's structure is entirely up to you and your team to create. In our application we'll use EJS templates, which are similar in structure to HTML. EJS is similar to PHP, JSP (for Java), and ERB (for Ruby), where server-side JavaScript is embedded in an HTML document and executed prior to sending to the client. We'll look at EJS more in depth in chapter 10. To inform `express(1)` that we want to use EJS templates, we can specify that with that `-e` flag.

```
wavded@dev:~/Projects/photo
.
├── app.js
├── package.json
└── public
    ├── images
    ├── javascripts
    └── stylesheets
        └── style.css
├── routes
    ├── index.js
    └── user.js
└── views
    └── index.ejs

6 directories, 6 files
[wavded@dev photo]$ _
```

Figure 9.4 Default application skeleton structure using EJS templates

By the end of this chapter you'll have an application with a similar but expanded structure, as shown in figure 9.5.

```
wavded@dev: ~/Projects/photo
.
├── app.js
├── models
│   └── Photo.js
├── package.json
└── public
    ├── images
    ├── javascripts
    └── photos
        └── style.css
├── routes
│   └── photos.js
└── views
    └── photos
        ├── index.ejs
        └── upload.ejs

9 directories, 7 files
wavded@dev ~/Projects/photo» _
```

Figure 9.5 Final application structure

In the next section you'll be:

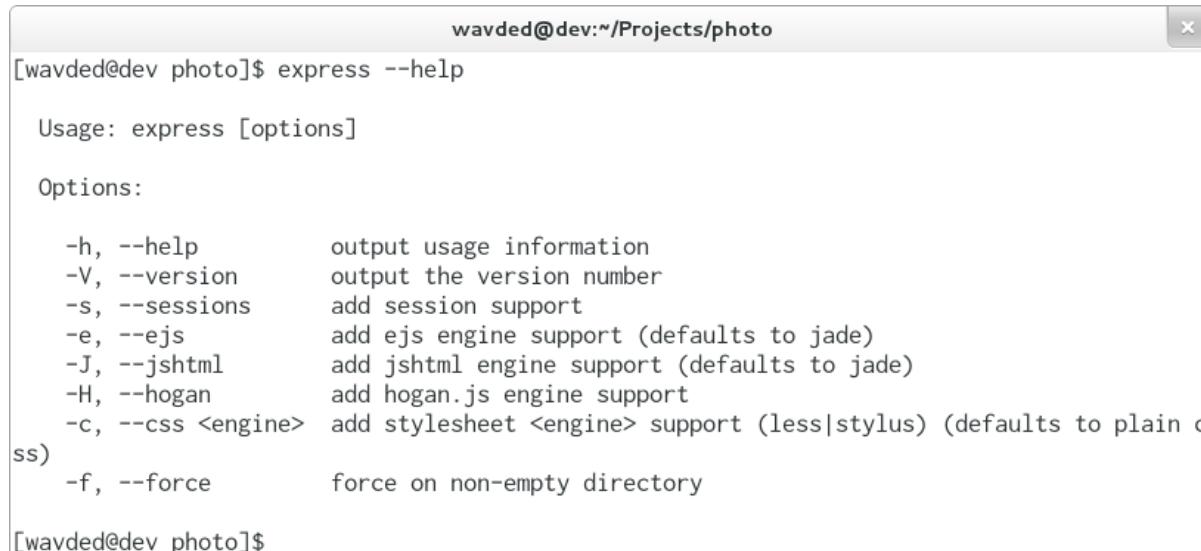
- Installing Express globally with npm
- Exploring the options provided by the executable
- Inspecting the generated application files
- Installing the initial application dependencies with npm
- Running the application for the first time

9.1.1 Installing the Express executable

First you'll want to install Express globally with npm to access the executable:

```
$ npm install -g express
```

Once installed, you can use the `--help` flag to see the options available, as shown in figure 9.6:



```
wavded@dev:~/Projects/photo
[wavded@dev photo]$ express --help
Usage: express [options]

Options:

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

[wavded@dev photo]$ _
```

Figure 9.6 Express help

Some of these options will generate small portions of the application for you. For example, you can specify a template engine and a dummy template file will be generated for the chosen template engine. Similarly, if you specify a CSS preprocessor using the `--css` option a dummy template file will be generated for it. If you use the `--sessions` option, session middleware will be enabled.

With the executable installed let's generate what will become the photo application.

9.1.2 Generating and exploring the application

For this application you'll only be using the `-e` (or `--ejs`) flag in order to use the EJS templating engine. Execute `express -e photo`. A fully functional application is then created in the `photo` directory, which contains a package.json file to describe the project and dependencies, the application file itself, the public file directories, and a directory for routes (see figure 9.7):

```
wavded@dev:~/Projects
[wavded@dev Projects]$ express -e photo

create : photo
create : photo/package.json
create : photo/app.js
create : photo/public
create : photo/public/javascripts
create : photo/public/images
create : photo/routes
create : photo/routes/index.js
create : photo/routes/user.js
create : photo/views
create : photo/views/index.ejs
create : photo/public/stylesheets
create : photo/public/stylesheets/style.css

install dependencies:
$ cd photo && npm install

run the app:
$ node app

[wavded@dev Projects]$ _
```

Figure 9.7 Generating the Express application

EXPLORING THE APPLICATION

Let's take a closer look at what was generated. Open the `package.json` file in your editor to see the application's dependencies, as shown in the following listing 9.8. Express can't guess which version of dependencies you'll want so it's good practice to supply the major, minor, and patch-levels of the module so you don't introduce any surprise bugs. For example, `"express": "3.0.0"` is explicit and will provide you with identical code on each installation.

```
wavded@dev:~/Projects/photo
[wavded@dev photo]$ cat package.json
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app"
  },
  "dependencies": {
    "express": "3.0.0",
    "ejs": "*"
  }
[wavded@dev photo]$ _
```

Figure 9.8 Generated package.json contents

To add the latest version of a module, in this case EJS, npm may be passed the

--save flag on installation. Execute the following command, then open package.json again to see the change:

```
$ npm install ejs --save
```

Now look at the application file generated by express(1), shown in listing 9.2. For now you'll leave this file as is. You should be familiar with these middleware from the Connect chapter, but it's worth taking a look at how the default middleware configuration is set up.

Listing 9.2 Generated Express application skeleton

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

var app = express();
app.configure(function(){
    app.set('port', process.env.PORT || 3000);
    app.set('views', __dirname + '/views');
    app.set('view engine', 'ejs');
    app.use(express.favicon()); ①
    app.use(express.logger('dev')); ②
    app.use(express.bodyParser()); ③
    app.use(express.methodOverride());
    app.use(app.router);
    app.use(express.static(path.join(__dirname, 'public'))); ④
});
app.configure('development', function(){
    app.use(express.errorHandler()); ⑤
});

app.get('/', routes.index); ⑥
app.get('/users', user.list);

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

- ① Serve default favicon
- ② Output development-friendly colored logs
- ③ Parse request bodies
- ④ Serve static files from ./public
- ⑤ Display styled HTML error pages in development
- ⑥ The application routes itself

You've seen the `package.json`, and the `app.js` file, but the application won't run yet because the dependencies haven't been installed. Whenever you generate a `package.json` file from `express(1)` you'll need to install the dependencies (as shown in figure 9.9). Execute `npm install` to do this step, then execute `node app` to fire up the application. Check out the application by visiting `localhost:3000` in your browser.



```
wavded@dev photo]$ npm install
ejs@0.8.3 node_modules/ejs

express@3.0.0 node_modules/express
└── methods@0.0.1
    ├── fresh@0.1.0
    ├── range-parser@0.0.4
    ├── cookie@0.0.4
    ├── crc@0.2.0
    ├── commander@0.6.1
    ├── debug@0.7.0
    ├── mkdirp@0.3.3
    └── send@0.1.0 (mime@1.2.6)
    └── connect@2.6.0 (pause@0.0.1, bytes@0.1.0, formidable@1.0.11, qs@0.5.1, send@0.0.4)
[wavded@dev photo]$ node app.js
Express server listening on port 3000
```

Figure 9.9 Install dependencies and run application

The default application generated looks like the following figure 9.10:

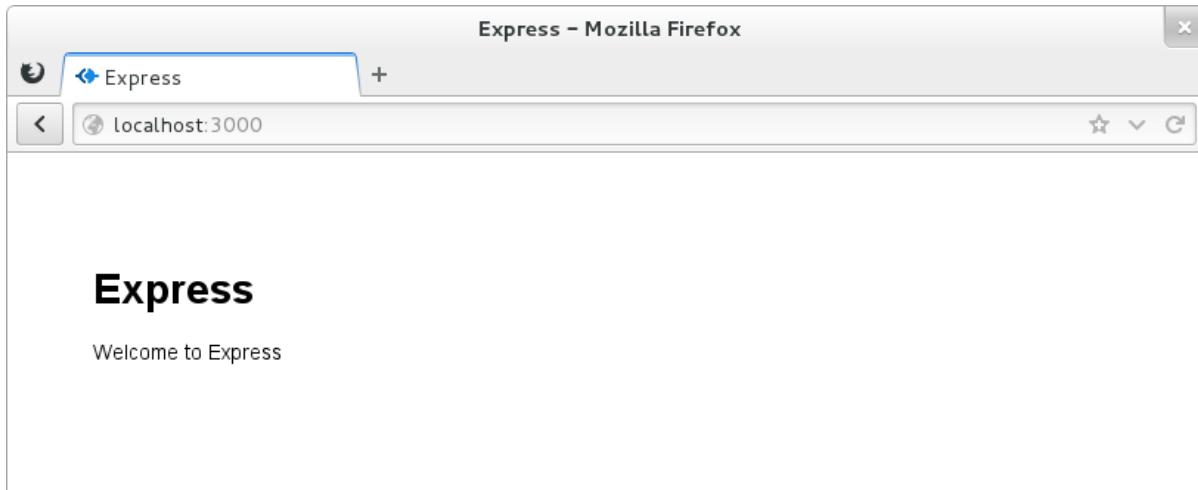


Figure 9.10 Default Express application

Now that you've seen the generated application let's dive into the environment-specific configuration.

9.2 Configuring Express and your application

Most of the time you have different requirements for your application depending on the environment in which it's running. For example, you may want to have verbose logging when your product's in development, but a leaner set of logs and gzip compression when it's in production. In addition to configuring environment-specific functionality, you may want to define some application-level settings so Express knows what template engine you're using and what folder in which to find the templates. Express also lets you define custom configuration key/value pairs.

The Express configuration system is a minimalistic environment-driven implementation, consisting of the following five methods, all driven by the `NODE_ENV` environment variable:

- `app.configure()`
- `app.set()`
- `app.get()`
- `app.enable()`
- `app.disable()`

SIDE BAR Setting environmental variables

To set an environmental variable in UNIX systems you can use:

```
$ NODE_ENV=production node app
```

In Windows you can use:

```
$ set NODE_ENV=production
$ node app
```

These environment variables will be available to you in your application on the `process.env` object.

In this section you'll see how to use the configuration system to customize how Express behaves, as well as how to use it for your own purposes throughout development. Let's take a closer look at what "environment-based configuration" means.

9.2.1 Environment-based configuration

Although the `NODE_ENV` environment variable originated in Express, many other Node frameworks have adopted it as a means to notify the application which environment it's operating within, defaulting to "development."

As shown in the following listing 9.3, the `app.configure()` method accepts optional strings representing the environment, and a function. When the environment matches the string passed, the callback is immediately invoked, and when only a function is given it will be invoked for all environments. These environment names are completely arbitrary, for example, you may have "development," "stage," "test," and "production" or "prod" for short.

Listing 9.3 Using the `app.configure()` to set configuration options that apply only to a development environment

```
app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'ejs');
  ...
});
app.configure('development', function(){
  app.use(express.errorHandler());
});
```

① All environments
② Development only

To illustrate that `app.configure()` is purely "sugar," the following listing 9.4 would be equivalent. You're not forced to use this feature; for example, you could load configuration from JSON or YAML.

Listing 9.4 Using a conditional statement to set configuration options that apply only to a development environment

```
var env = process.env.NODE_ENV || 'development'; ①
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs'); ②
...
if ('development' == env) { ③
  app.use(express.errorHandler());
}
```

① Default to "development"

② All environments

③ Development only using “if” statement instead of app.configure

Express uses the configuration system internally, allowing you to customize how Express behaves, but it’s also available for your own use. For this application you’ll only be using a single setting “photos,” which is the directory that will be used to store the uploaded images, as you see in the following code. This value could be changed in “production” to save and serve photos from an different volume with more disk space. To retrieve this value you would then invoke `app.get('photos')`.

```
app.configure(function(){
  ...
  app.set('photos', __dirname + '/public/photos');
  ...
});

app.configure('production', function(){
  ...
  app.set('photos', '/mounted-volume/photos');
  ...
});
```

Express also provides boolean variants of `app.set()` and `app.get()`. For example, `app.enable(setting)` is equivalent to `app.set(setting, true)`, and `app.enabled(setting)` may be used to check if the value was enabled, and the methods `app.disable(setting)` and `app.disabled(setting)` complement the truthful variants.

Now that you’ve seen how to take advantage of the configuration system for your own use, let’s look at rendering views in Express.

9.3 Rendering views

In our application we’ll utilize EJS templates, though as previously mentioned almost any template engine in the Node community may be used. If you’re not familiar with EJS don’t worry. It’s similar to templating languages found in other languages (PHP, JSP, ERB). We’ll cover some basics of EJS in this chapter, but we’ll discuss EJS and several others in greater detail in chapter 10.

Whether it’s rendering an entire HTML page, an HTML fragment, or an RSS feed, rendering views is crucial for nearly every application. The concept is simple: you pass data to a “view,” and that data is transformed, typically to HTML for web applications. You’re likely familiar with the idea of views, as most frameworks

provide similar functionality, but just in case, the following figure [xref] illustrates how a view forms a new representation for the data:

```
{ name: 'Tobi', species: 'ferret', age: 2 }
```



```
<h1>Tobi</h1>
<p>Tobi is a 2 year old ferret.</p>
```

Figure 9.11 HTML template plus data = HTML view of data

Express provides two ways to render views: at the application-level with `app.render()`, and at the request- or response-level with `res.render()`, which uses the former internally. In this chapter, you'll only be using `res.render()`. If you look in `./routes/index.js` there's a single function exported: the `index` function. This function invokes `res.render()` in order to render the `./views/index.ejs` template, as shown in the following code:

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

In this section you'll see how to:

- configure the Express view system
- look up view files
- find the various techniques you can use to expose data to them when rendering views

Before looking at `res.render()` more closely let's configure the view system.

9.3.1 View system configuration

Configuring the Express view system is simple, and although `express(1)` generated the configuration for you, it's still useful to know what's going on behind the scenes so you can make changes. You'll see how:

- to adjust the view lookup
- to configure your default template engine
- to enable view caching to reduce file I/O

First up is the “views” setting.

CHANGING THE LOOKUP DIRECTORY

The following snippet shows the “views” setting that the Express executable created.

```
app.set('views', __dirname + '/views');
```

This setting is the directory that Express will use during view lookup, defaulting to `./views`. But it's a good idea to use `__dirname` so that your application isn't dependent on the current working directory being the application's root.

SIDE BAR

`__dirname`

`__dirname` (note the two leading underscores) is a global directory in Node in which the currently running file exists. Many times in development this directory will be the same as your current working directory (CWD), but in production the Node executable may run from another directory (using `__dirname` helps keep paths consistent across environments).

The next setting is “view engine,” an optional setting that allows you to omit the template engine extension.

DEFAULT TEMPLATE ENGINE

The “view engine” setting was assigned to “ejs” for you when `express(1)` generated the application because EJS was the template engine selected by the `-e` command-line option. This is the setting that enables you to render “index,” rather than “index.ejs.” otherwise Express requires the extension in order to determine which template engine is to be used.

You might be wondering why Express even considers extensions. This feature allows you to use multiple template engines within a single Express application, while providing a clean API for the common use-cases, because most applications will use one template engine for the majority of the application.

Suppose, for example, you find writing RSS feeds easier with another template engine, or perhaps you’re migrating from one template engine to another. You might use Jade as the default, and EJS for the “/feed” route, as indicated in the following snippet by the “.ejs” extension:

Listing 9.5 A template filename’s extension names the template engine that should be used to render the template

```
app.set('view engine', 'jade');
app.get('/', function(){
    res.render('index'); ①
});
app.get('/feed', function(){
    res.render('rss.ejs') ②;
});
```

- ① .jade assumed because it’s set as a ‘view engine’
- ② Because the .ejs extension is provided use the EJS template engine

SIDE BAR Keep in mind that any additional template engines you wish to use should be added to your package.json dependencies object.

VIEW CACHING

The “view cache” setting is enabled by default in the “production” environment and prevents subsequent `render()` calls from performing disk I/O. The contents of the templates are saved in-memory, greatly improving performance. The side-effect of enabling this setting is that you may no longer edit the template files without restarting the server, which is why it’s disabled in development.

If you’re running a “stage” or a “staging” environment, you’ll likely want to enable this option.

As illustrated in the following figure (9.12), when view cache is disabled the template is read from disk on every request. This is what allows you to make changes to a template without restarting the application. When “view cache” is enabled the disk is only hit once per template.

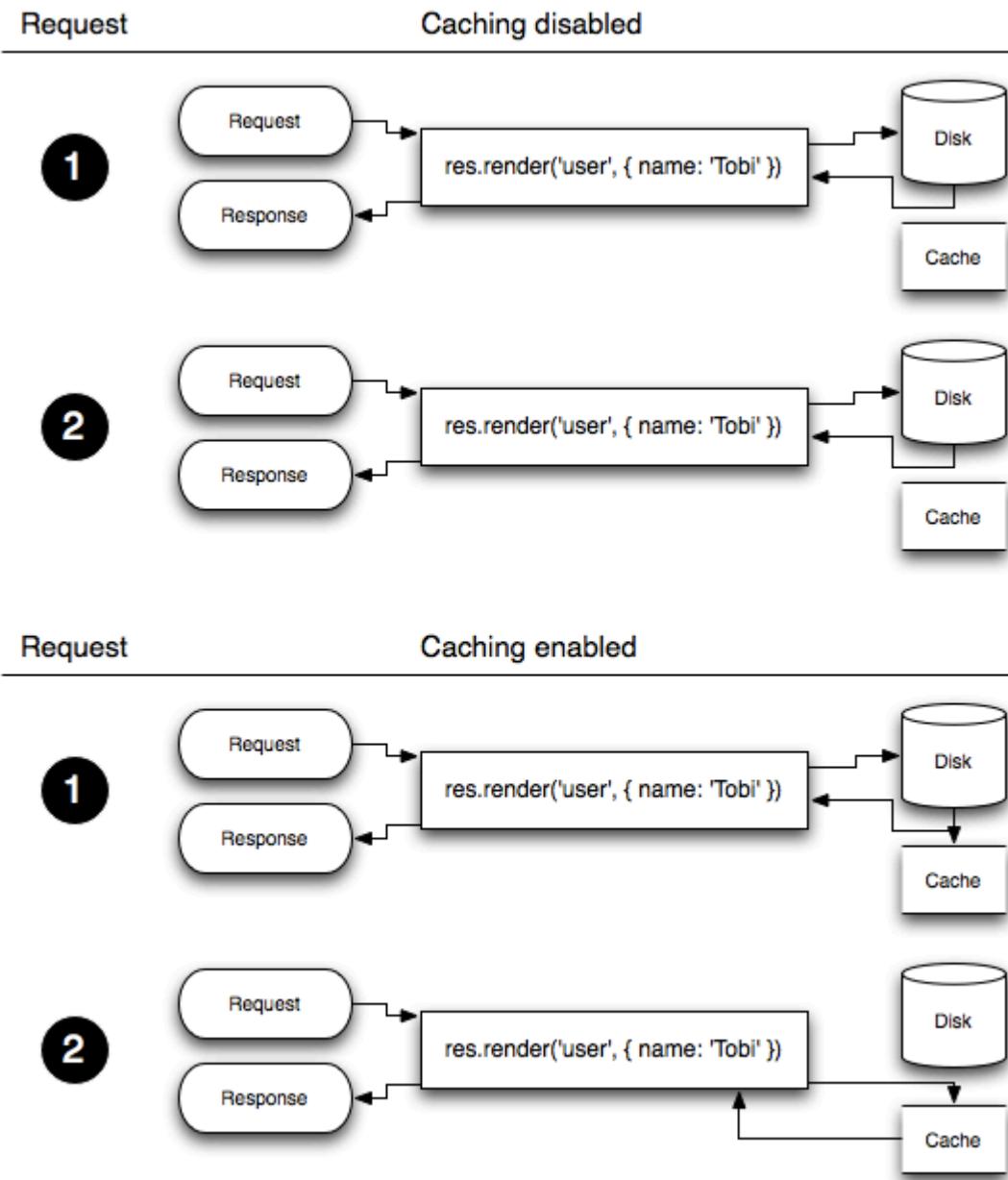


Figure 9.12 The “view cache” setting

You’ve seen how the view caching mechanism helps improve performance when in a non-development environment, now let’s see how Express locates views in order to render them.

9.3.2 View lookup

Now that you know how to configure the view system let’s take a look at how it resolves where the target view file is located, the “view lookup” (see figure 9.14). Don’t worry about creating these templates yet, you’ll be doing this later.

This process is similar to how Node’s `require()` works. When

`res.render()` or `app.render()` are invoked Express will first check if a file exists with an absolute path. Next, it will look relative to the “views” directory setting discussed in section 9.2 “Configuring Express” section and, finally, Express will try an `index` file. This process is represented as a flowchart in figure 9.13.

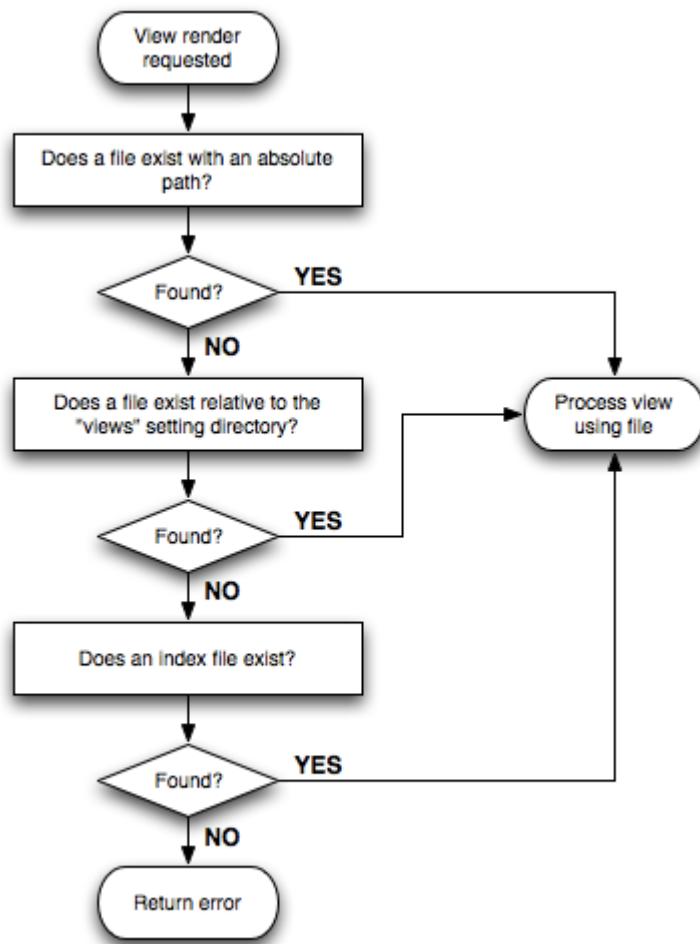


Figure 9.13 Express view lookup process

Because “ejs” is set as the default engine the render call omits the “.ejs” extension, but is still resolved correctly. As the application evolves you’ll need more views, sometimes several for a single resource.

Using view lookup can help with organization—for example, you can use sub-directories related to the resource and create views within them, as illustrated by the `photos` directory in figure 9.14.

```
wavded@dev:~/Projects/photo
views
└── index.ejs
└── layout.ejs
photos
└── index.ejs
└── upload.ejs
1 directory, 4 files
[wavded@dev photo]$
```

Figure 9.14 Express view lookup

Adding the directory allows you to eliminate redundant names such as “upload-photo.ejs” and “show-photo.ejs.” Express will then add the “view engine” extension and resolve the view as ./views/photos/upload.ejs.

Express will check to see if a file named “index” resides in that directory. When used with a pluralized resource such as “photos” this typically implies a listing. This is illustrated in `res.render('photos')` in figure 9.14.

Now that you know how Express looks up views let’s start creating the photo listings and put this feature to work.

9.3.3 Exposing data to views

Express provides several mechanisms to expose local variables to the views being rendered, but first you need something to render. In this section you’ll be using some “dummy” data to populate the initial photo listing view.

Before getting databases involved let’s create this placeholder data. Create a file named `./routes/photos.js`, which will contain the photo-specific routes. For now, create a `photos` array in this same file that will act as the faux database, as the following code shows:

Listing 9.6 Dummy photo data you’ll use to populate the view

```
var photos = [];
photos.push({
  name: 'Node.js Logo',
  path: 'http://nodejs.org/images/logos/nodejs-green.png'
});
photos.push({
  name: 'Ryan Speaking',
  path: 'http://nodejs.org/images/ryan-speaker.jpg'
});
```

Now that you have some content, you'll need a route to display them.

CREATING THE PHOTO LISTING VIEW

To start displaying the dummy photo data you'll be defining a route that will render an EJS “photos” view, as shown in figure 9.15, which you'll be creating next.

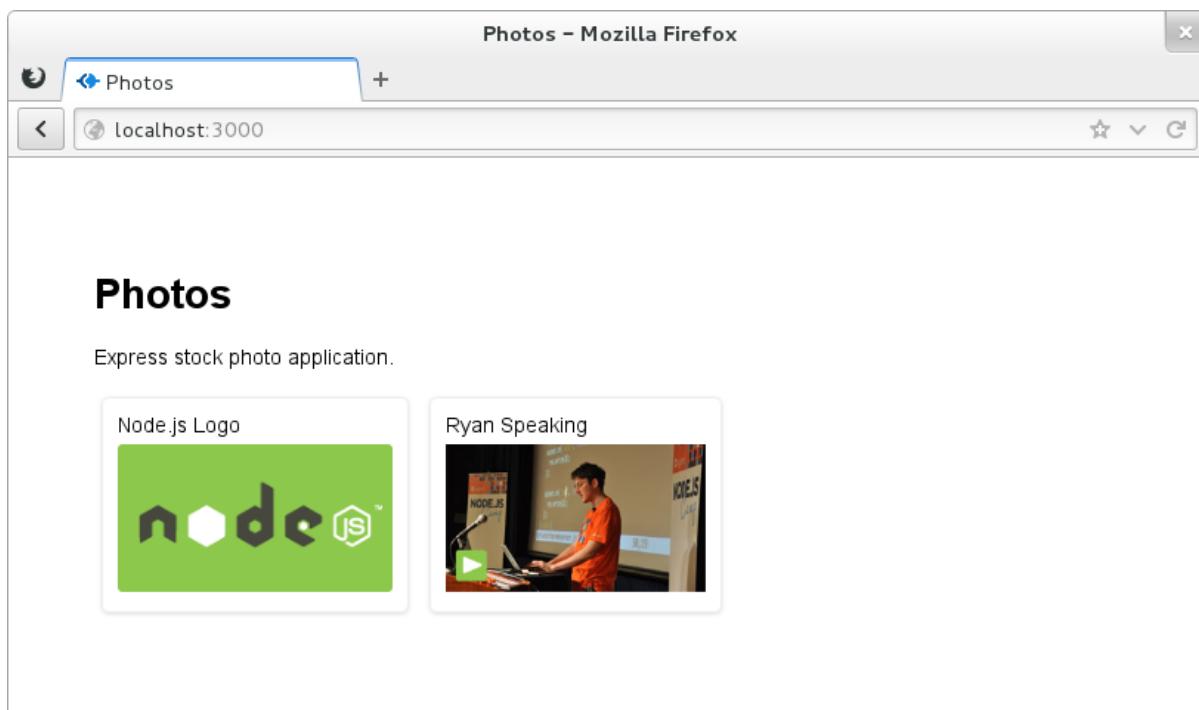


Figure 9.15 Initial photo listing view

To get started, open up `./routes/photos.js` and export a function named “lis” (see the following code). In practice this can be named whatever you like. Route functions are identical to regular Connect middleware functions, accepting a request and response object, as well as the `next()` callback, which isn't used in this example, so it's omitted. This is the first and main method of passing objects to a view, by passing an object to `res.render()`.

Listing 9.7 List route

```
exports.list = function(req, res){
  res.render('photos', {
    title: 'Photos',
    photos: photos
  });
};
```

In `./app.js` you may then require the `photos` module to get access to the `exports.list` function you previously wrote. To display the photos for the index page “`/`”, pass the `photos.list` function to the `app.get()` method, which is used to map the HTTP method GET, and the path matching “`/`” to this function.

Listing 9.8 Adding photos.list route

```
...
, routes = require('../routes')
, photos = require('../routes/photos');
...
app.get('/', photos.list);
```

**① replaces `app.get('/'`,
`routes.index)`**

With the dummy data and route set up you can write the photo view. You’ll have several photo-related views, so create a directory named `./views/photos` and `index.ejs` inside of it. Using a JavaScript `forEach` you can then iterate each photo in the `photos` object that was passed to `res.render()`. Each photo name and image is then displayed, as the following code shows:

Listing 9.9 A view template to list photos

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title> ①
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1>Photos</h1>
    <p>Express stock photo application.</p>
    <div id="photos">
      <% photos.forEach(function(photo) { %> ②
        <div class="photo">
          <h2><%=photo.name%></h2>
          <img src='<%=photo.path%>' />
        </div>
      <% }) %>
    </div>
  </body>
</html>
```

- ① EJS outputs escaped values by using <%= value %>
- ② EJS executes vanilla JS using <% code %>

This view will produce markup similar to the following listing 9.10 :

Listing 9.10 HTML produced by the photos/index.ejs template

```
...
<h1>Photos</h1>
<p>Express stock photo application.</p>
<div id="photos">
  <div class="photo">
    <h2>Node.js Logo</h2>
    
  </div>
...
...
```

If you're interested in styling your application, to follow is the CSS used for ./public/stylesheets/style.css:

Listing 9.11 CSS to style this chapter's tutorial application

```
body {
  padding: 50px;
  font: 14px "Helvetica Neue", Helvetica, Arial, sans-serif;
}
a { color: #00B7FF; }
.photo {
  display: inline-block;
  margin: 5px;
  padding: 10px;
  border: 1px solid #eee;
  border-radius: 5px;
  box-shadow: 0 1px 2px #ddd;
}
.photo h2 {
  margin: 0;
  margin-bottom: 5px;
  font-size: 14px;
  font-weight: 200;
}
.photo img { height: 100px; }
```

Fire up the application with node app, and take a look at

`http://localhost:3000` in your browser and you'll have the photo display shown in figure 9.15.

METHODS OF EXPOSING DATA TO VIEWS

You've seen how you can pass local variables directly to `res.render()` calls, but you can also use a few other mechanisms for this. For example, `app.locals` may be used for application-level variables and `res.locals` to store request-level local variables.

The values passed directly to `res.render()` will take precedence over values set in `res.locals` and `app.locals`, as figure 9.16 shows.

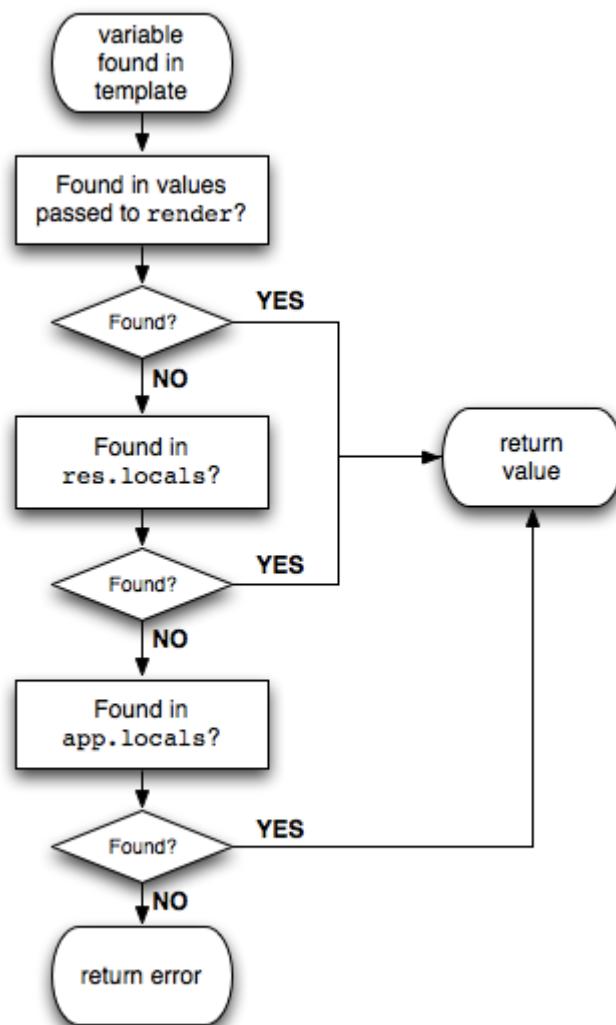


Figure 9.16 Values passed directly to the `render` function take precedence when rendering a template

By default Express exposes only one application-level variable, “`settings`,”

which is the object containing all of the values set with `app.set()`. For example, suppose you have `app.set('title', 'My Application')`, you could access this in the template as shown in the following EJS snippet:

```
<html>
  <head>
    <title><%=settings.title%></title>
  </head>
  <body>
    <h1><%=settings.title%></h1>
    <p>Welcome to <%=settings.title%>.</p>
  </body>
```

Internally, Express exposes this object with the following JavaScript (and that's all there is to it):

```
app.locals.settings = app.settings;
```

For convenience, `app.locals` is also a JavaScript function. When an object is passed, all the keys will be merged, so if you have existing objects that you wish to expose in their entirety such as some i18n data, you may do the following:

```
var i18n = {
  prev: 'Prev',
  next: 'Next',
  save: 'Save'
};
app.locals(i18n);
```

This will expose `prev`, `next`, and `save` to all templates. This feature works great to expose view helpers to encourage less logic within templates. For example, suppose you have the Node module “`helpers.js`” with a few functions exported, you could expose all of these to the views by doing the following:

```
app.locals(require('./helpers'));
```

Let's add a way to upload files unto this site and learn how Express uses

Connect's `bodyParser` middleware to make that possible.

9.4 Handling forms and file uploads

Let's implement the photo upload feature. Make sure you have the "photos" setting defined for this application, as discussed earlier in the configuration section. This will give you the freedom to change the directory in various environments and will make things simpler if you change your mind. But for now, they'll be saved in `./public/photos`, as the following code shows:

Listing 9.12 A custom application setting allows you to set a destination for uploaded photos.

```
...
app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'ejs');
  app.set('photos', __dirname + '/public/photos');
...
}
```

The first step in implementing the photo upload feature is to define the photo model to support photo interaction. We'll do that in the next section, as well as:

- create a photo upload form
- implement a photo listing

9.4.1 Implementing the Photo model

We'll use a simple Mongoose model we learned about in section 5.3.3 of the Database chapter to make our model. Install Mongoose with `npm install mongoose --save`. Then, create the file `./models/Photo.js` with our model definition shown in listing 9.13.

Listing 9.13 .A model for our photos

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/photo_app'); ①
var schema = new mongoose.Schema({
  name: String,
  path: String
});
module.exports = mongoose.model('Photo', schema);
```

- ① Set up connection to mongodb on localhost and use ‘photo_app’ as database

Mongoose provides all the CRUD methods (`Photo.create`, `Photo.update`, `Photo.remove`, `Photo.find`) already on the model, so we’re done.

9.4.2 Creating a photo upload form

With the Photo model implemented you can now implement the upload form and related routes. Much like the other pages you’ll need a GET route and a POST route for the uploads page. In the case of the POST handler you’ll be passing the photos directory to it, returning a route callback so it has access to it. Add the new routes to `app.js` below the default (/) route:

```
...
app.get('/upload', photos.form);
app.post('/upload', photos.submit(app.get('photos')));
...
```

CREATING THE PHOTO UPLOAD FORM

Next you’ll be creating the upload form shown in the following figure 9.17. This form contains an optional photo name and a file input for the image.

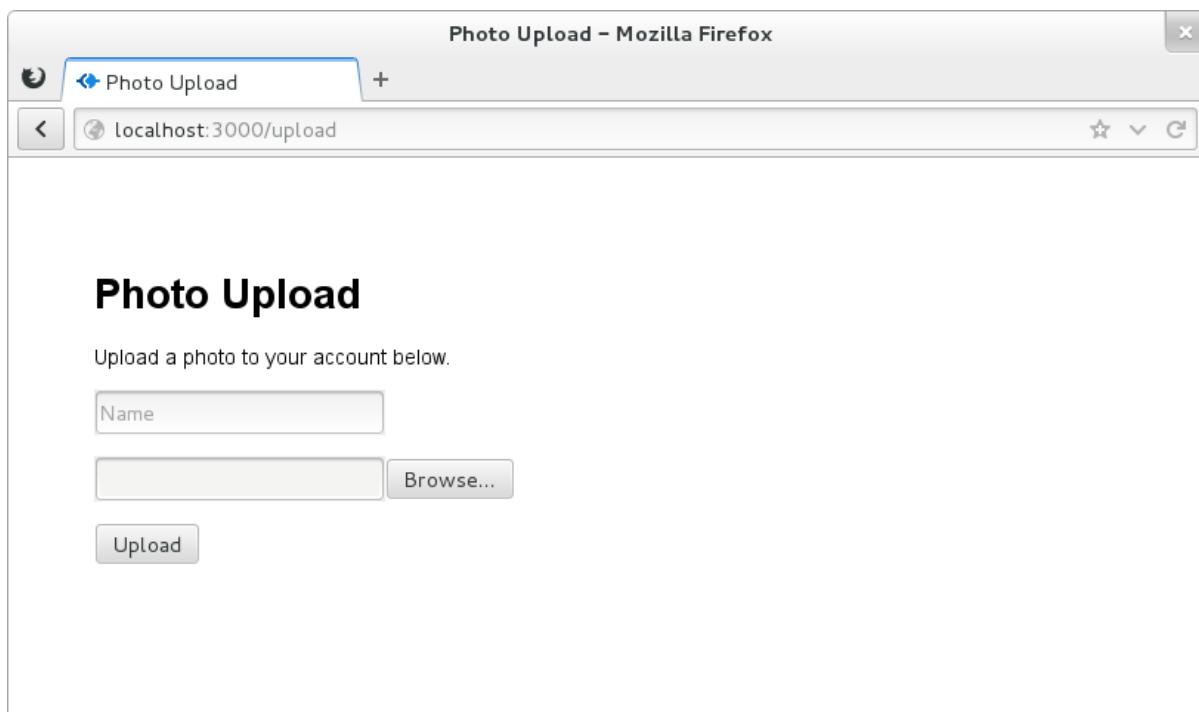


Figure 9.17 Photo upload form

Create `./views/photos/upload.ejs` with the following EJS code to produce the form:

Listing 9.14 A form for uploading photos

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Upload a photo to your account below.</p>
    <form method='post' enctype='multipart/form-data'>
      <p><input type='text', name='photo[name]', placeholder='Name' />
      </p>
      <p><input type='file', name='photo[image]' /></p>
      <p><input type='submit', value='Upload' /></p>
    </form>
  </body>
</html>
```

ADDING A ROUTE FOR THE PHOTO UPLOAD PAGE

Now you have the form, but no way to display it. The `photos.form` function will do this. In `./routes/photos.js` export the `form` function, which will render `./views/photos/upload.ejs`:

Listing 9.15 Add the form route

```
exports.form = function(req, res){
  res.render('photos/upload', {
    title: 'Photo upload'
  });
};
```

HANDLING PHOTO SUBMISSIONS

Next you'll need a route to handle the form submission. As discussed in the Connect chapter, the `bodyParser()` and specifically the `multipart()` middleware (which `bodyParser` includes) will provide you with a `req.files` object representing files that have been uploaded and saved to disk. Because the field name given in the upload form is “photo[image]” you can access this object via `req.files.photo.image`, and `req.body.photo.name` for “photo[name].”

The file is “moved” with `fs.rename()` to its new destination within the `dir` passed to `exports.submit()`. Remember in your case this is the “photos” setting you defined in “app.js.” A new `Photo` object is populated and saved. Upon success the user is redirected to the index page, as the following code shows:

Listing 9.16 Adding photo submit route definition

```

var Photo = require('../models/Photo');
var path = require('path');
var fs = require('fs');
var join = path.join;
...
exports.submit = function (dir) {
  return function(req, res, next){
    var img = req.files.photo.image;
    var name = req.body.photo.name || img.name;
    var path = join(dir, img.name);
    fs.rename(img.path, path, function(err){
      if (err) return next(err);
      Photo.create({
        name: name,
        path: img.name
      }, function (err) {
        if (err) return next(err);
        res.redirect('/');
      });
    });
  };
};

```

1 Require the Photo model

2 Reference “path.join” so we can name variables “path”

3 Default to original file name

4 Rename the file

5 Delegate errors

6 Delegate errors

7 Perform an HTTP redirect to the index page

Great! Now you can upload photos. Next, you’ll be implementing the logic necessary to display them on the index page.

9.4.3 Showing a list of uploaded photos

In “Exposing data to views” section you implemented the route `app.get('/photos.list')` using dummy data. Now it’s time for the real thing.

Previously, the route callback did little more than pass the dummy array of photos to the template, as shown in the following code:

```

exports.list = function(req, res){
  res.render('photos', {
    title: 'Photos',
    photos: photos
  });
};

```

The updated version uses `Photo.find`, provided in Mongoose, to grab every photo you ever uploaded. Later you’ll be implementing pagination as this example will perform poorly with a large collection of photos.

Once the callback is invoked with an array of photos the reset of the route remains the same as it was before introducing the asynchronous query.

Listing 9.17 Modified list route

```
exports.list = function(req, res, next){
  Photo.find({}, function(err, photos){
    if (err) return next(err);
    res.render('photos', {
      title: 'Photos',
      photos: photos
    });
  });
};
```

① {} means find all records in photo collection

Let's also update our `./views/photos/index.ejs` template to be relative to our `./public/photos`, rather than any URL.

Listing 9.18 Modified view to use settings for photos path

```
...
<% photos.forEach(function(photo) { %>
  <div class="photo">
    <h2><%=photo.name%></h2>
    <img src='/photos/<%=photo.path%>' />
  </div>
<% } ) %>
...
```

The index page will now display a dynamic list of photos uploaded through the application, as shown in the following figure 9.18:

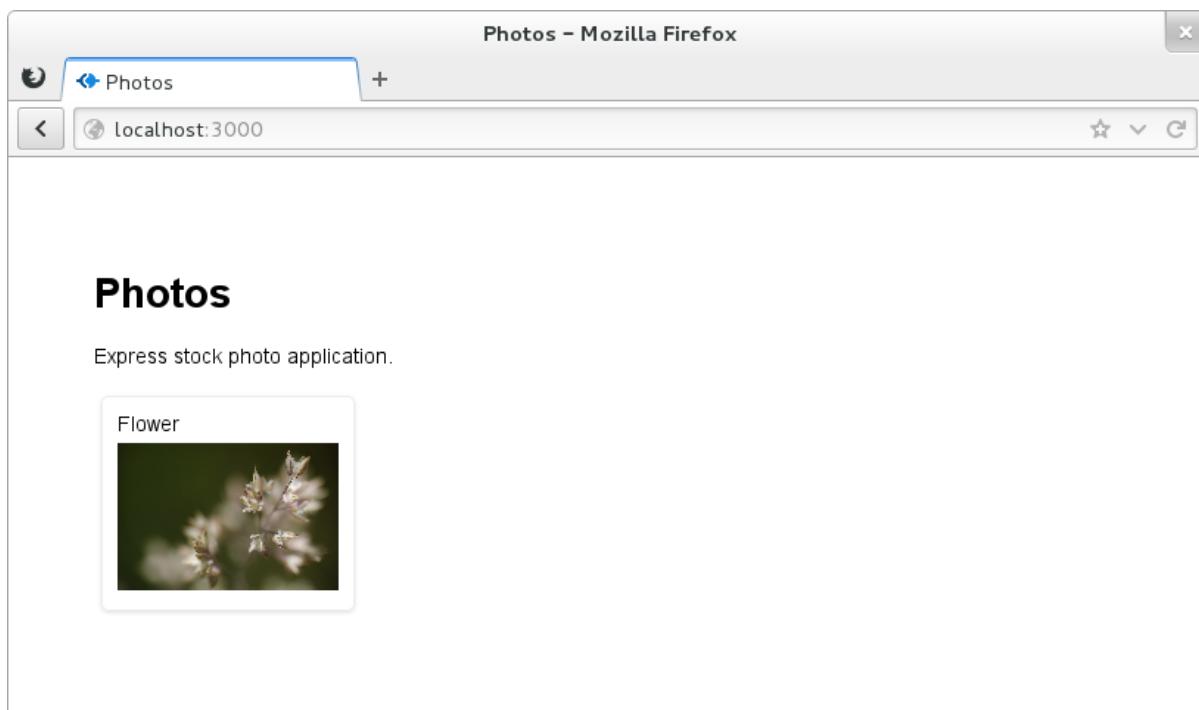


Figure 9.18 The photo application as built to this point

So far the routes you've defined have been somewhat simple: they don't accept wildcards, and no validation is performed on the routes. So let's dive into the routing capabilities of Express.

9.5 Handling resource downloads

Up until now you've only been serving static files with the `express.static()` middleware. Express, however, provides several helpful response methods for dealing with file transfers. These include `res.sendFile()` to transfer files and the `res.download()` variant, which prompts the browser to save the file.

In this section, you'll be tweaking the application so that original photos can be downloaded by adding a "GET /photo/:id/download" route.

9.5.1 Creating the photo download route

First things first, you'll need to add a link to the photos so that users may download them. Open up `./views/photos/index.ejs` and change it to the following listing. This change adds a link around the `img` tag pointing to the "GET /photo/:id/download" route.

Listing 9.19 Add a download hyperlink to download

```
...
<% photos.forEach(function(photo) { %>
  <div class="photo">
    <h2><%=photo.name%></h2>
    <a href="/photo/<%=photo.id%>/download">
      <img src='/photos/<%=photo.path%>'/>
    </a>
  </div>
<% } ) %>
...

```

1 Mongoose
provides an ID field
that can be used to
look up a specific
record

Back in `app.js`, define the following route anywhere you like amongst the others.

```
app.get('/photo/:id/download', photos.download(app.get('photos')));
```

Before you can try this out, you need the download route. Let's implement it.

9.5.2 Implementing the photo download route

In `./routes/photos.js` export a “download” function as shown in listing 9.20. This route loads up the requested photo and transfers the file at that given path. `res.sendFile()` is provided by Express, and is backed by the same code as `express.static()`, so you get http cache, range, and other features for free. Consequently this method takes the same options. For example you may pass `{ maxAge: oneYear }` as the second argument.

Listing 9.20 Photo download route

```
exports.download = function(dir){ ①
  return function(req, res, next){ ②
    var id = req.params.id;
    Photo.findById(id, function(err, photo){ ③
      if (err) return next(err);
      var path = join(dir, photo.path); ④
      res.sendfile(path); ⑤
    });
  };
}
```

- ① The directory from which to serve files
- ② The route callback itself
- ③ Load the photo record
- ④ Construct an absolute path to the file
- ⑤ Transfer the file

NOTE

A callback may also be provided as the second or third (when using options) argument to notify when a download is complete. For example, you could use this to decrement a user's credits.

If you fire up the application you should now be able to click the photos when authenticated. The result you get may not be what you'd expected. With `res.sendFile()` the data is transferred and interpreted by the browser. In the case of images, the browser will display them within the window as shown in figure 9.19. Next we'll look at `res.download()`, which is used to prompt the browser for download.

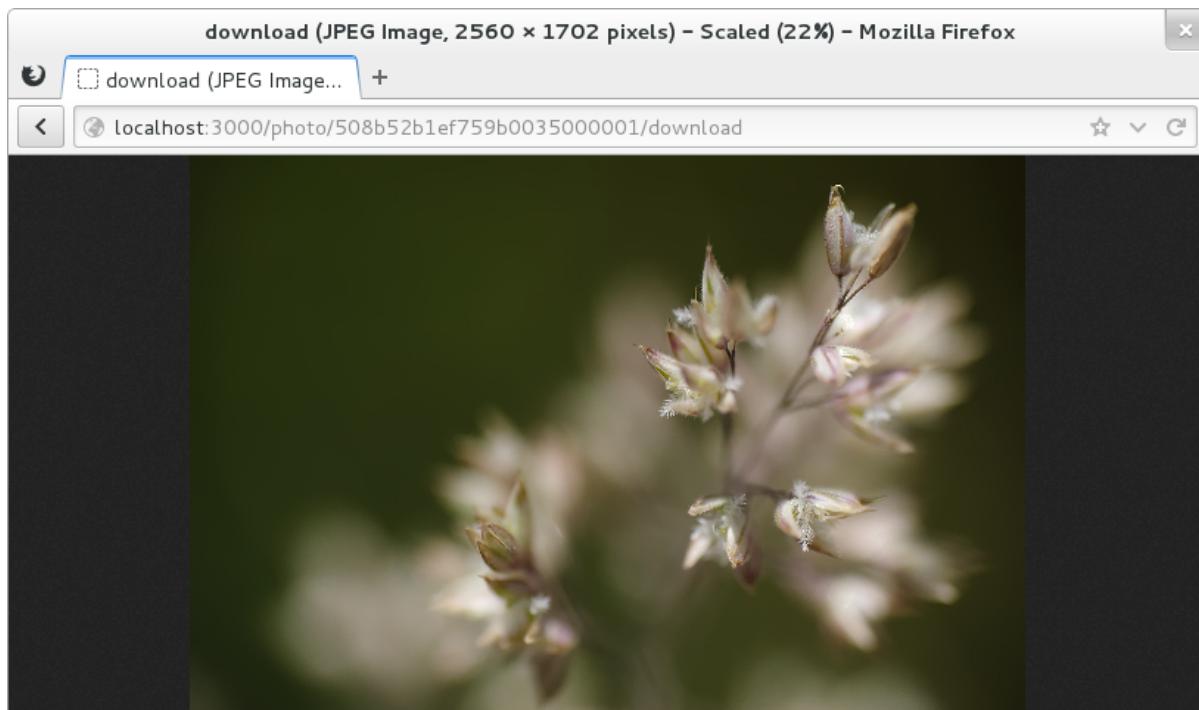


Figure 9.19 Photo transferred with `res.sendFile()`

TRIGGER A BROWSER DOWNLOAD

When you switch `res.sendFile()` with `res.download()`, it will alter the behavior of browsers when files are transferred. The “Content-Disposition” header field is set to the file’s basename and the browser will prompt for download accordingly. Figure 9.20 shows how the original image’s basename of “`littlenice_by_dhor.jpeg`” was used. Depending on your application this might not be ideal. Let’s look at `res.download()`’s optional filename argument next.

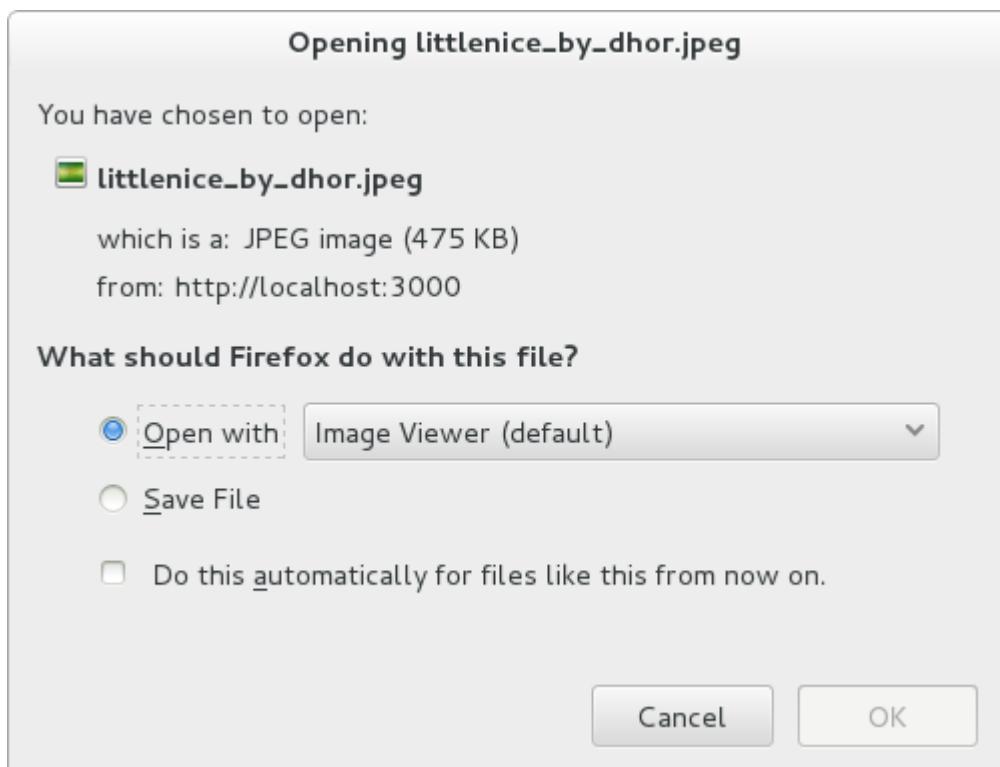


Figure 9.20 Photo transferred with `res.download()`

SETTING A DOWNLOAD'S FILENAME

The second argument of `res.download()` allows you to define an explicit filename to be used rather than defaulting to the original. Listing 9.21 changes the previous implementation to provide the name given when uploaded, such as “Flower.jpeg.”

Listing 9.21 Photo download route with explicit filename

```
...
var path = join(dir, photo.path);
res.download(path, photo.name+'.jpeg');
...
```

If you fire up the application and try clicking a photo now you should be prompted to download a file similar to the one shown in the following figure:

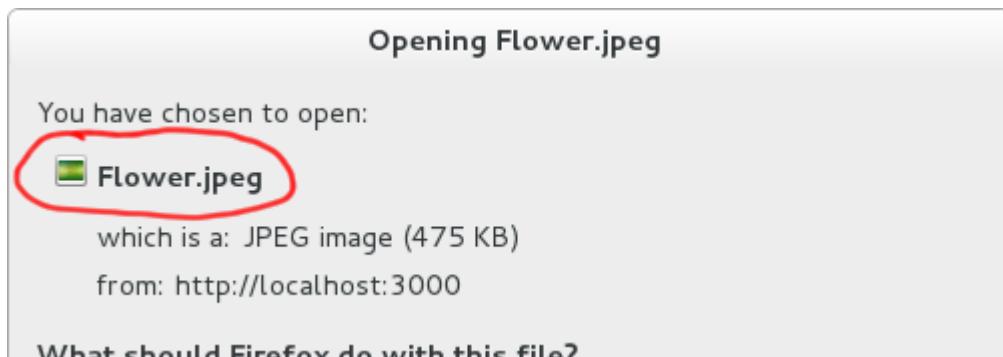


Figure 9.21 Photo transferred with `res.download()` and a custom filename

9.6 Summary

In this chapter you learned how to create an Express application from scratch and how to deal with common web development needs.

You learned how a typical Express application's directories are structured and how to use environmental variables and the `app.configure` method to change application behavior for different environments.

The most primary components of Express applications are routes and views. You learned how to render views and how to expose data to them by setting `app.locals`, `res.locals`, and by passing values directly using `res.render()`. We also covered how basic routing works.

In the next chapter we'll go into more advanced techniques you can do with Express, such as authentication, routing and middleware techniques, and REST APIs.

10

Web application templating

In this chapter:

- How templating helps keep applications organized
- Creating templates using Embedded JavaScript
- Learning minimalist templating with Hogan
- Using Jade to create templates

In chapter 9 you learned some templating basics to use with the Express framework in order to create views. In this chapter you'll focus on templating exclusively, learning how to use three popular template engines and how to use templating to keep any web application's code clean by separating logic from presentation markup. If you're familiar with templating and the model-view-controller (MVC) pattern, you can skim through to section 10.2, where you'll start learning about the template engines we'll be detailing in this chapter, which include Embedded JavaScript, Hogan, and Jade. If you're not familiar with templating, keep reading—we'll explore it conceptually in the next few sections.

10.1 Using templating to keep code clean

You can use the model–view–controller pattern to develop conventional web applications in Node as well as in nearly every other web technology. One of the key concepts to MVC is the separation of logic, data and presentation. In MVC web applications, the user will typically request a resource from the server, which will cause the “controller” to request application data from the “model”, and then pass the data to the “view”, which will finally format the data for the end user. This “view” portion is often implemented using one of various templating languages. When an application uses templating, the view will relay selected values, returned by the model, to a “template engine,” and specify what “template” file should be used to define how to display the provided values.

Figure 10.1 shows how templating logic fits into the overall architecture of an MVC application:

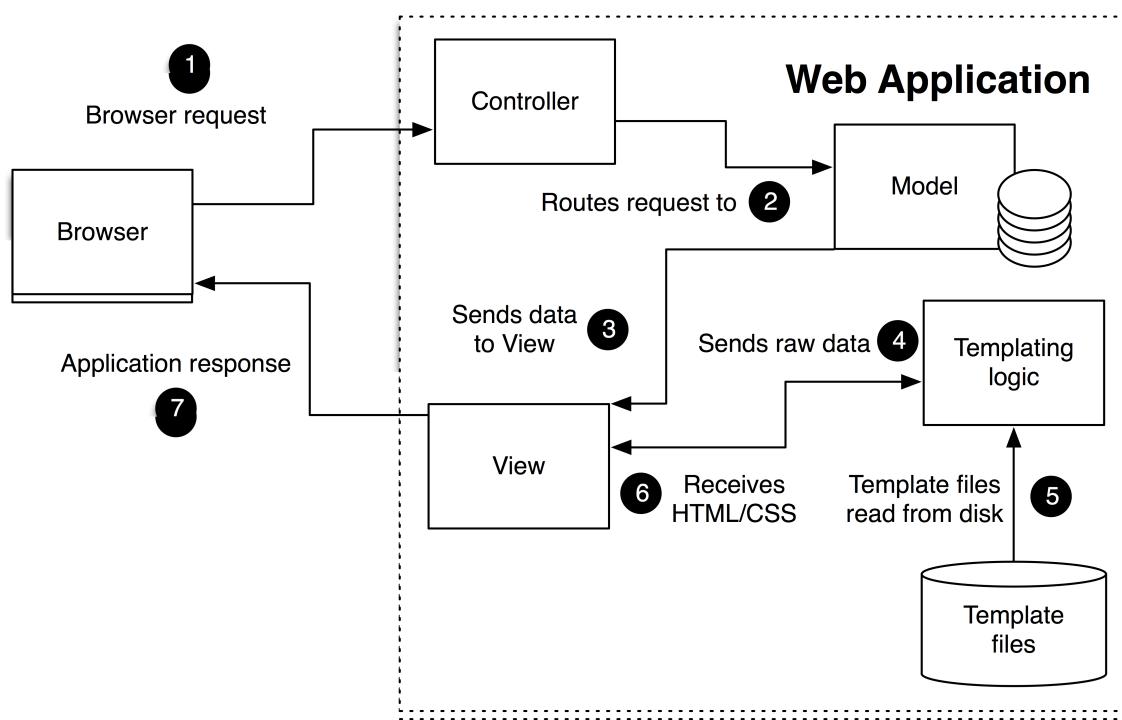


Figure 10.1 The flow of an MVC application and its interaction with the template layer

Template files typically contain placeholders for application values as well as HTML, CSS, and sometimes small bits of client-side JavaScript to do things like display third-party widgets, such as the Facebook “like” button, or trigger interface

behavior, such as hiding or revealing parts of the page. As template files focus on presentation rather than logic, front-end developers can work on them, in addition to server-side developers, which can help with a project's division of labor.

In this section, we'll render HTML with, and without, a template to show you the difference. But first, let's start with an example of templating in action.

10.1.1 Templating in action

As a quick illustration of how you can apply templating, let's look at the problem of elegantly outputting HTML from a simple blogging application. Each blog entry will have a title, date of entry, and body text. The blog will look similar to what's shown in figure 10.2, in a web browser:

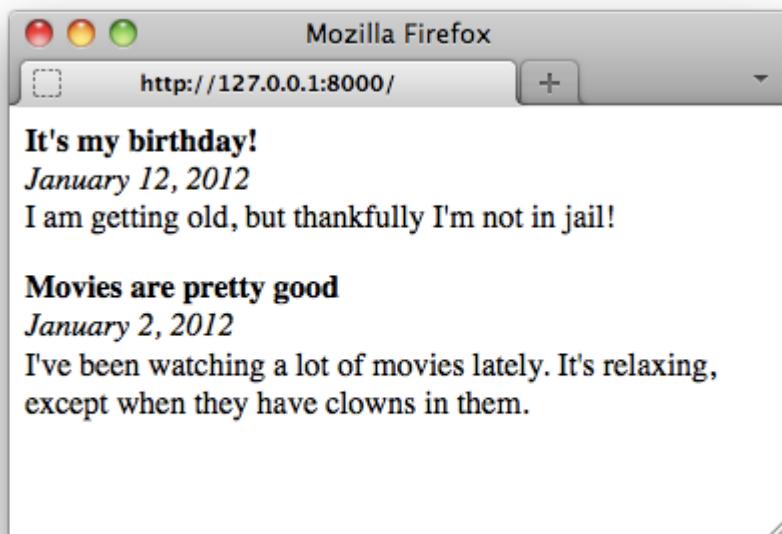


Figure 10.2 Example blog application browser output

Blog entries will be read from a text file formatted like the following snippet. The --- in listing 10.1 indicates where one entry stops and another begins:

Listing 10.1 entries.txt: Blog entries text file

```
title: It's my birthday!
date: January 12, 2012
I am getting old, but thankfully I'm not in jail!
---
title: Movies are pretty good
date: January 2, 2012
I've been watching a lot of movies lately. It's relaxing,
```

```
except when they have clowns in them.
```

The blog application code will start by requiring necessary modules and reading in the blog entries, as shown in listing 10.2:

Listing 10.2 blog.js: Blog entry file parsing logic of a simple blogging application

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

function getEntries() { ①
  var entries = [];
  var entriesRaw = fs.readFileSync('../entries.txt', 'utf8'); ②

  entriesRaw = entriesRaw.split("---"); ③

  entriesRaw.map(function(entryRaw) {
    var entry = {};
    var lines = entryRaw.split("\n"); ④

    lines.map(function(line) { ⑤
      if (line.indexOf('title: ') === 0) {
        entry.title = line.replace('title: ', '');
      }
      else if (line.indexOf('date: ') === 0) {
        entry.date = line.replace('date: ', '');
      }
      else {
        entry.body = entry.body || '';
        entry.body += line;
      }
    });
    entries.push(entry);
  });
  return entries;
}

var entries = getEntries();
console.log(entries);
```

- ① Function to read and parse blog entry text
- ② Read blog entry data from file
- ③ Parse text into individual blog entries
- ④ Parse entry text into individual lines

 Parse lines into entry properties

The following code, when added to the blog application, defines an HTTP server. When the server receives an HTTP request it'll return a page containing all blog entries. This page is rendered using a function called `blogPage`, which we'll define next.

```
var server = http.createServer(function(req, res) {
  var output = blogPage(entries);

  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end(output);
});

server.listen(8000);
```

Now, define the `blogPage` function, which renders the blog entries into a page of HTML that can be sent to the user's browser. You'll implement this trying two approaches:

- Rendering HTML without a template
- Rendering HTML using a template

10.1.2 Rendering HTML without a template

The blog application could output the HTML directly, but including the HTML with the application logic would result in clutter. In listing 10.3, the `blogPage` function illustrates a non-templated approach to displaying blog entries:

Listing 10.3 blog_manual.js: Logic gets cluttered with presentation details when a template engine isn't employed

```
function blogPage(entries) {
  var output = '<html>
    + '<head>
      + '<style type="text/css">
        + '.entry_title { font-weight: bold; }'
        + '.entry_date { font-style: italic; }'
        + '.entry_body { margin-bottom: 1em; }'
      + '</style>
    + '</head>'
```

➊ Too much HTML interspersed with logic

```

+ '<body>';

entries.map(function(entry) {
  output += '<div class="entry_title">' + entry.title + "</div>\n"
  + '<div class="entry_date">' + entry.date + "</div>\n"
  + '<div class="entry_body">' + entry.body + "</div>\n";
});

output += '</body></html>';

return output;
}

```

Note that all of this presentation-related content, CSS definitions, and HTML, adds many lines to the application.

10.1.3 Rendering HTML using a template

Implementing HTML using templating allows you to remove the HTML from the application logic, cleaning up the code considerably.

To try out the demos in this section, you'll need to install the "ejs" module into your application directory. You can do this by entering the following into the command line:

```
npm install ejs
```

The following code loads a template from a file then defines a new version of the `blogPage` function, this time using the Embedded JavaScript (EJS) template engine, which we'll show you how to use in section 10.2.

```

var ejs = require('ejs');
var template = fs.readFileSync('../template/blog_page.ejs', 'utf8');

function blogPage(entries) {
  var values = {entries: entries};
  return ejs.render(template, {locals: values});
}

```

The EJS template file would contain HTML markup (keeping it out of the application logic) and placeholders that indicate where data passed to the template

engine should be put. The EJS template file that shows the blog entries would contain the HTML and placeholders shown in the following listing 10.4:

Listing 10.4 blog_page.ejs: An EJS template for displaying blog entries

```
<html>
  <head>
    <style type="text/css">
      .entry_title { font-weight: bold; }
      .entry_date { font-style: italic; }
      .entry_body { margin-bottom: 1em; }
    </style>
  </head>

  <body>
    <% entries.map(function(entry) { %> ①
      <div class="entry_title"><%= entry.title %></div> ②
      <div class="entry_date"><%= entry.date %></div>
      <div class="entry_body"><%= entry.body %></div>
    <% }); %>
  </body>
</html>
```

- ① Placeholder that loops through blog entries
- ② Placeholders for bits of data in each entry

Community-contributed Node modules provide template engines, and a wide variety of them exist. If you consider HTML and/or CSS inelegant, because HTML requires closing tags and CSS requires opening and closing braces, take a closer look at template engines. They allow template files to use special “languages” (such as the Jade language, which we’ll cover later in this chapter) that provide a “shorthand” way of specifying HTML and/or CSS. These template engines can make your templates cleaner, but you may not want to take the time to learn an alternative way of specifying HTML and CSS. Ultimately, what you decide to use comes down to a matter of personal preference.

In the rest of this chapter, you’ll learn how to incorporate templating in your Node applications through the lens of three popular template engines:

- the Embedded JavaScript (EJS) engine
- the minimalist Hogan engine
- the template engine Jade

Each of these engines allow you to write HTML in an alternative way. Let's start with EJS.

10.2 Templating with Embedded JavaScript

Embedded JavaScript¹ (simply "EJS" for short) takes a fairly straightforward approach to templating and will be familiar territory for folks who've used template engines in other languages, such as JSP (Java), Smarty (PHP), ERB (Ruby), etc. EJS allows you to embed "EJS tags" as placeholders for data within HTML. EJS also lets you execute raw JavaScript logic in your templates, in a way similar to PHP, for tasks such as conditional branching and iteration.

Footnote 1 <https://github.com/visionmedia/ejs>

In this section you'll learn how:

- To create EJS templates
- EJS filters can provide commonly needed, presentation-related functionality such as text manipulation, sorting, and iteration
- To integrate EJS in your Node applications
- To use EJS for client-side applications

10.2.1 Creating a template

In the world of templating the data sent to the template engine for rendering is sometimes called the "context." The following is a bare-bones example of Node using EJS to render a simple template using a context:

```
var ejs = require('ejs');
var template = '<%= message %>';
var context = {message: 'Hello template!'};

console.log(ejs.render(template, {locals: context}));
```

Note the use of `locals` in the second argument sent to `render`. The second argument can include rendering options as well as context data, which means the use of `locals` ensures that individual bits of context data aren't interpreted as EJS options. But it's possible in most cases to pass the context itself as the second option, as the following `render` call illustrates:

```
console.log(ejs.render(template, context));
```

If you pass a context in directly as the second argument to `render`, make sure you don't name context values using any of the following:

“cache”	“filename”	“scope”
“debug”	“compileDebug”	“client”
“open”	“close”	

These values are reserved to allow the changing of template engine settings.

CHARACTER ESCAPING

When rendering, EJS escapes any special characters in context values, replacing them with HTML entity codes. You do this to prevent cross-site scripting attacks (XSS)² in which malicious web application users attempt to submit JavaScript as data in the hopes that, when displayed, it'll execute in some other user's browser. The following code shows EJS's escaping at work:

Footnote 2 https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29

```
var ejs = require('ejs');
var template = '<%= message %>';
var context = {message: "<script>alert('XSS attack!');</script>"};

console.log(ejs.render(template, context));
```

The previous code will display the following output:

```
&lt;script&ampgtalert('XSS attack!');&lt;/script&ampgt
```

If you trust data being used in your template and don't want to escape a context value in an EJS template you can use `<%-` instead of `<%=` in your template tag, as the following code demonstrates:

```
var ejs = require('ejs');
var template = '<%- message %>';
var context = {
  message: "<script>alert('Trusted JavaScript!');</script>"
}

console.log(ejs.render(template, context));
```

Note that if you don't like the characters used by EJS to specify tags, you can customize this, as the following example shows:

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

ejs.open = '{{' 
ejs.close = '}}'

var template = '{= message }';
var context = {message: 'Hello template!'};

console.log(ejs.render(template, context));
```

Now that you know the basics of EJS, let's look at things you can do with it that'll make managing presentation of data easier.

10.2.2 Manipulating template data using EJS filters

EJS provides support for “filters”—a feature that allows you to easily do lightweight data transformations without code. When using filters add a “`:`” to the opening characters of your EJS tag. For example:

- `<%=:`: would be used for escaped EJS output
- `<%-:` for unescaped output

Filters can also be “chained,” meaning you can put multiple filters in a single EJS tag and display the cumulative effect of all filters (similar to the “pipe” concept on *NIX systems). In the next few sections we'll run through a number of

filters that are useful in common scenarios.

FILTERS THAT HANDLE SELECTION

EJS filters are put into EJS tags. To give you an example of the usefulness of filters, imagine an application that allows users to let people know what movies they've watched. One bit of important information might be the most recent movie they've watched. The EJS tag in the template in the following example displays the last movie in an array of movies by using a `last` filter to display only the last item in an array:

```
var ejs = require('ejs');
var template = '<%= movies | last %>';
var context = {'movies': [
  'Bambi',
  'Babe: Pig in the City',
  'Enter the Void'
]};

console.log(ejs.render(template, context));
```

Note that `first` is also a valid filter. If you want to get a specific item in the list you could use the `get` filter. The EJS tag `<%= movies | get:1 %>` would display the second item in the `movies` array (with item 0 being the first item). You can also use the `get` filter to show properties if the context value is an object rather than an array.

FILTERS FOR CASE MANIPULATION

EJS filters can also be used to change case. The EJS tag in the following template includes a filter that'll capitalize the first letter in a context value, in this case changing the displayed value from “bob” to “Bob”:

```
var ejs = require('ejs');
var template = '<%= name | capitalize %>';
var context = {name: 'bob'};

console.log(ejs.render(template, context));
```

If you want to display a context value entirely to upper case you could use the `upcase` filter. Conversely, using the `downcase` filter would display the value in

lower case.

FILTERS FOR TEXT MANIPULATION

Text can be sliced and diced by EJS filters. You can truncate text, append to prepended text, and even replace parts of your text.

Truncating text to a certain character count allows you to eliminate the problem of long strings of text causing problems with HTML layouts. The following code, for example, will truncate the title text to 20 characters, displaying “The Hills are Alive”:

```
var ejs = require('ejs');
var template = '<%= title | truncate:20 %>';
var context = {title: 'The Hills are Alive With the Sound of Critters'};

console.log(ejs.render(template, context));
```

If you want to truncate text to a certain number of words, an EJS filter supports that, too. In the previous example you could replace the EJS tag with <%= title | truncate_words:2 %> to truncate the context value to two words. The output would then be “The Hills”.

The “replace” filter uses `String.prototype.replace(pattern)` behind the scenes, so it accepts either a string or a RegExp. The following code shows an example of automatically abbreviating a word using an EJS filter:

```
var ejs = require('ejs');
var template = "<%= weight | replace:'kilogram','kg' %>";
var context = {weight: '40 kilogram'};

console.log(ejs.render(template, context));
```

You can append text by adding a filter like `append:'some text'`. Similarly, you can prepend text using a filter like `prepend:'some text'`.

FILTERS THAT DO SORTING

EJS filters can also sort. Returning to the previously cited movie title example, you could use EJS filters to sort the movies by title and display the first item in alphabetical order, as illustrated by figure 10.3:

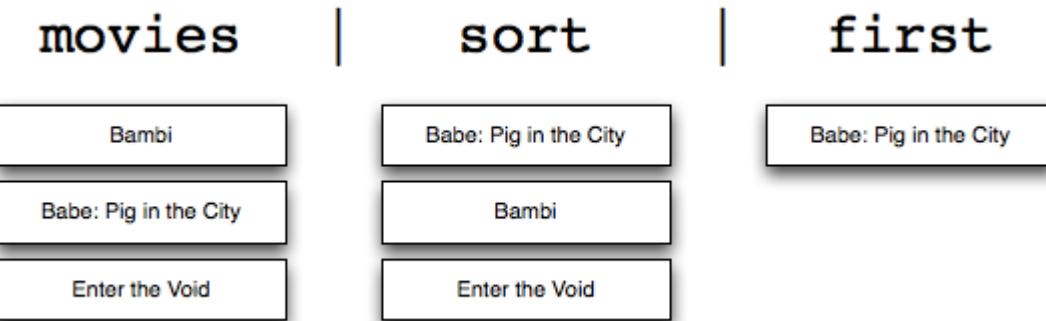


Figure 10.3 Visualizing the use of EJS filters to process arrays of text

The following code would implement this:

```
var ejs = require('ejs');
var template = '<%= movies | sort | first %>';
var context = {'movies': [
  'Bambi',
  'Babe: Pig in the City',
  'Enter the Void'
]};

console.log(ejs.render(template, context));
```

If you want to sort an array composed of objects, but you'd like to sort by comparing object properties, you can do this using filters, as the following code shows:

```
var ejs = require('ejs');
var template = "<%= movies | sort_by:'name' | first | get:'name' %>";
var context = {'movies': [
  {name: 'Babe: Pig in the City'},
  {name: 'Bambi'},
  {name: 'Enter the Void'}
]};

console.log(ejs.render(template, context));
```

Note the use of `get : 'name'` at the end of the filter chain. You use it because what the sort returns is an object and you need to select what property of the object to display.

THE "MAP" FILTER

The EJS map filter allows you to specify the property of an object that you want subsequent filters to operate on. In the previous example we could use the filter chain using `map`. As an alternative to having to specify the property using the `sort_by` filter and having to specify the property to display using the `get` filter, you would use the `map` filter to create an array from object properties. The resulting EJS would be `<%=: movies | map: 'name' | sort | first %>`.

CREATING CUSTOM FILTERS

Although EJS comes with filters for most common needs, you may want something beyond what EJS offers. If you want a filter that could, for example, round to an arbitrary decimal place, you'd find there's no built-in filter to do this. Luckily, with EJS it's easy to add your own custom filters, as listing 10.5 shows:

Listing 10.5 custom_ejs.js: Defining your own custom EJS filters

```
var ejs = require('ejs');
var template = '<%=: price * 1.14 | round:2 %>';
var context = {price: 21};

ejs.filters.round = function(number, decimalPlaces) { ①
  number = isNaN(number) ? 0 : number; ②
  decimalPlaces = !decimalPlaces ? 0 : decimalPlaces;

  var multiple = Math.pow(10, decimalPlaces);
  return Math.round(number * multiple) / multiple;
}

console.log(ejs.render(template, context));
```

- ① Simply define a function on the `ejs.filters` object
- ② The first argument is the input value the context or previous filter result

As you can see, filters in EJS provide a great way to lessen the amount of logic you need to prepare data for display. Rather than doing these transformations to your data manually before rendering the template, EJS provides nice built-in mechanisms for doing that for you.

10.2.3 Integrating EJS in your application

Because it's awkward to store templates in files along with application code and doing this clutters up your code, we'll show you how to use Node's filesystem API to read them from separate files.

Move to a working directory and create a file named `app.js` containing the code in listing 10.6:

Listing 10.6 app.js: Storing template code in files

```

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

var filename = './template/students.ejs'; ①

var students = [ ②
  {name: 'Rick LaRue', age: 23},
  {name: 'Sarah Cathands', age: 25},
  {name: 'Bob Dobbs', age: 37}
];

var server = http.createServer(function(req, res) { ③
  if (req.url == '/') {
    fs.readFile(filename, function(err, data) { ④
      var template = data.toString();
      var context = {students: students};
      var output = ejs.render(template, context); ⑤
      res.setHeader('Content-type', 'text/html');
      res.end(output); ⑥
    });
  } else {
    res.statusCode = 404;
    res.end('Not found');
  }
});

server.listen(8000);

```

- ① Note location of template file
- ② Data to pass to template engine
- ③ Create HTTP server
- ④ Read template from file
- ⑤ Render template
- ⑥ Send HTTP response

Next, create a child directory in it called `template`. In this directory you'll keep your templates. Create a file named `template/students.ejs` in the `template` directory and enter the code in listing 10.7 into it:

Listing 10.7 `students.ejs`: EJS template that renders an array of "students"

```
<% if (students.length) { %>
<ul>
  <% students.forEach(function(student) { %>
    <li><%= student.name %> (<%= student.age %>)</li>
  <% } %>
</ul>
<% } %>;
```

CACHING EJS TEMPLATES

EJS supports optional, in-memory caching of template functions. What this means is that EJS, after parsing your template file once, will store the function that's created by the parsing. Rendering a cached template will be faster because the parsing step can be skipped.

If you're doing initial development of a Node web application, and want to see any changes you make to your template files reflected immediately, you won't want to enable caching. But if you're deploying an application to production, enabling caching is a quick, easy win. Notice how caching is conditionally enabled via the `NODE_ENV` environment variable.

To try out caching, change the call to EJS's `render` function in the previous example to the following:

```
var cache = process.env.NODE_ENV === 'production';
var output = ejs.render(
  template,
  {students: students, cache: cache, filename: filename}
);
```

Note that the `filename` option doesn't necessarily have to be a file—you can use a unique value that identifies whichever template you're rendering.

Now that you've learned how to integrate EJS with your Node applications, let's look at how EJS can be used in a different way: in web browsers.

10.2.4 Using EJS for client-side applications

We've shown an example that uses EJS with Node; now let's now take a quick look at using EJS in the browser. To use EJS on the client-side you'll first want to download the EJS engine to your working directory, as shown by the following commands:

```
cd /your/working/directory
curl https://raw.github.com/visionmedia/ejs/master/ejs.js -o ejs.js
```

If you don't have `curl` by now then you can download the file at <https://github.com/visionmedia/ejs/downloads> in your web browser. Once you download the `ejs.js` file, then you can use EJS in your client-side code. The following listing 10.8 shows a simple client-side application of EJS:

Listing 10.8 browser_ejs.html: Using EJS to add templating capabilities to the client-side

```
<html>
  <head>
    <title>EJS example</title>
    <script src="ejs.js"></script>
    <script ①
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.8/jquery.js">
    </script>
  </head>
  <body>

    <div id='output'></div> ②

    <script>
      var template = "<%= message %>"; ③
      var context = {message: 'Hello template!' }; ④

      $(document).ready(function() {
        $('#output').html(
          ejs.render(template, context) ⑤
        );
      });
    </script>
  </body>
```

```
</html>
```

- ① Include jQuery library for DOM manipulation
- ② Placeholder for rendered template output
- ③ Template to use to render content
- ④ Data to use with template
- ⑤ Wait until browser loads page
- ⑥ Render template to DIV with ID ‘output’

You’ve learned how to use a fully featured Node template engine, so it’s time to look at the Hogan template engine, which deliberately limits the functionality available to templating code.

10.3 Using the Mustache templating language with Hogan

Hogan.js³ is a template engine that was created by Twitter for its templating needs. Hogan is an implementation of the popular Mustache⁴ template language standard, which was created by GitHub’s Chris Wanstrath. Mustache takes a minimalist approach to templating. Unlike EJS, the Mustache standard deliberately doesn’t include conditional logic, nor any built-in content filtering capabilities other than escaping content to prevent XSS attacks. Mustache advocates that template code should be kept as simple as possible.

Footnote 3 <https://github.com/twitter/hogan.js>

Footnote 4 <http://mustache.github.com/>

In this section you’ll learn:

- How to create and implement Mustache templates in your application.
- The different template tags available in the Mustache standard.
- How to organize your templates using “partials.”
- How to fine-tune Hogan with your own delimiters and other options.

10.3.1 Creating a template

To use Hogan in an application, or to try out the demos in this section, you’ll need to install Hogan in your application directory. You can do this by entering the following into the command line:

```
npm install hogan.js
```

The following is a bare-bones example of Node using Hogan to render a simple template using a context. Running it will output the text “Hello template!”:

```
var hogan = require('hogan.js');
var template = '{{message}}';
var context = {message: 'Hello template!'};

var template = hogan.compile(template);
console.log(template.render(context));
```

Now that you know how to process Mustache templates with Hogan, let’s look at what tags Mustache supports.

10.3.2 Mustache tags

Mustache tags are conceptually similar to EJS’s tags. Mustache tags serve as placeholders for variable values, indicate where iteration is needed, and allow you to augment Mustache’s functionality and add comments to your templates.

DISPLAYING SIMPLE VALUES

To display a context value in a Mustache template, include the name of the value in double braces. Braces, in the Mustache community, are known as “mustaches.”

If you want to display the value for context item “name,” for example, you’d use the Hogan tag `{ { name } }`.

Like most template engines, Hogan escapes content by default to prevent XSS attacks. But to display an unescaped value in Hogan, you can either add a third mustache or prepend the name of the context item with an ampersand. Using the previous “name” example, you could display the context value unescaped by either using the `{ { { name } } }` or `{ { &name } }` tag formats.

If you want to add a comment in a Mustache template, you can use this format: `{ { ! This is a comment } }`.

SECTIONS: ITERATING THROUGH MULTIPLE VALUES

Although Hogan doesn’t allow the inclusion of logic in templates, it does include an elegant way to iterate through multiple values in a context item using Mustache “sections.”

The following context, for example, contains an item with an array of values:

```
var context = {
  students: [
    { name: 'Jane Narwhal', age: 21 },
    { name: 'Rick LaRue', age: 26 }
  ]
};
```

If you want to create a template that would display each student in a separate HTML paragraph, with output similar to the following, it would be a straightforward task using a Hogan template:

```
<p>Name: Jane Narwhal, Age: 21 years old</p>
<p>Name: Rick LaRue, Age: 26 years old</p>
```

The following template would produce the desired HTML:

```
{{#students}}
  <p>Name: {{name}}, Age: {{age}} years old</p>
{{/students}}
```

INVERTED SECTIONS: DEFAULT HTML WHEN VALUES DON'T EXIST

What if the value of the “students” item in the context data wasn’t an array? If the value was a single object, for example, the template would display it. But sections won’t display if the corresponding item’s value is undefined, false, or is an empty array. If you want your template to output a message indicating that values don’t exist for a section, Hogan supports what Mustache calls “inverted sections.” The following template code, if added to the previous student display template, would display a message when no student data exists in the context:

```
{{^students}}
  <p>No students found.</p>
{{/students}}
```

SECTION LAMBDA: CUSTOM FUNCTIONALITY IN SECTION BLOCKS

In order to allow developers to augment Mustache's functionality, the Mustache standard lets you define section tags that process template content rather than iterate through arrays.

As an example use of a section lambda, listing 10.9 shows how you'd use a lambda to add Markdown support when rendering a template. Note that the example uses the `github-flavored-markdown` module, which you'll have to install by entering `npm install github-flavored-markdown` into your command line.

In listing 10.9, the `**Name**` in the template gets rendered to `Name` when passing through the Markdown parser called by the lambda logic:

Listing 10.9 hogan_lambda.js: Using a lambda in Hogan

```
var hogan = require('hogan.js');
var md = require('github-flavored-markdown'); ①

var template = '{{{#markdown}}}
  + '**Name**: {{name}}
  + {{{/markdown}}}; ②

var context = {
  name:      'Rick LaRue',
  markdown: function() {
    return function(text) {
      return md.parse(text); ③
    };
  }
};

var template = hogan.compile(template);
console.log(template.render(context));
```

- ① Require Markdown parser
- ② Mustache template also contains Markdown formatting
- ③ The template context includes a lambda to parse Markdown in the template

Lambdas allow you to easily implement things like caching and translation mechanisms into your templates.

PARTIALS: REUSING TEMPLATES WITHIN OTHER TEMPLATES

When writing templates you want to avoid unnecessarily repeating the same code in multiple templates. One way to avoid this is to create “partials.” Partials are templates used as building blocks and included in other templates. Another use of partials is to break up complicated templates into simpler templates.

Listing 10.10, for example, uses a partial to separate the template code used to display student data from the main template:

Listing 10.10 hogan_partials: The use of partials in Hogan

```
var hogan = require('hogan.js');

var studentTemplate = '<p>Name: {{name}}, '1
    + 'Age: {{age}} years old</p>';
var mainTemplate = '{{#students}}'2
    + '{{>student}}'
    + '{{/students}}';

var context = {
  students: [
    {
      name: 'Jane Narwhal',
      age: 21
    },
    {
      name: 'Rick LaRue',
      age: 26
    }
  ]
};

var template = hogan.compile(mainTemplate);3
var partial = hogan.compile(studentTemplate);

var html = template.render(context, {student: partial});4
console.log(html);
```

- 1 Template code used for partial
- 2 Main template code
- 3 Compiling the main and partial templates
- 4 Rendering the main template and partial

10.3.3 Fine-tuning Hogan

Hogan is fairly simple to use—once you've learned its vocabulary of tags you should be off and running. You may need to tweak only a couple of things as you use it.

If you don't like Mustache-style braces, you can change the delimiters Hogan uses by passing the `compile` method an option to override them. The following example shows compiling in Hogan using EJS-style delimiters.

```
hogan.compile(text, {delimiters: '<% %>'});
```

If you'd like to use section tags that don't begin with the character '#' after the opening mustaches, you can do that with another `compile` method option: `sectionTags`. You might, for example, want to use a different tag format for section tags in which lambdas are employed. The following listing 10.11 alters the earlier example in listing 10.9 to use an underscore prefix to differentiate the `markdown` section tag from subsequent section tags that iterate rather than employ lambdas:

Listing 10.11 hogan_custom: Using custom section tags in Hogan

```
var hogan = require('hogan.js');
var md = require('github-flavored-markdown');

var template = '{ {_markdown} }'
  + '**Name**: { name }'
  + '{ /markdown }';

var context = {
  name:      'Rick LaRue',
  _markdown: function(text) {
    return md.parse(text);
  }
};

var template = hogan.compile(
  template,
  {sectionTags: [{o: '_markdown', c: 'markdown'}]});
console.log(template.render(context));
```

- ➊ Require Markdown parser
- ➋ Custom tag used in template
- ➌ Lambda for custom tag
- ➍ Custom opening and closing tags defined

When using Hogan, you won't have to change any options to enable caching. Caching is built into the `compile` function and enabled by default.

Now that you've learned two fairly straightforward Node template engines, let's look at the Jade template engine, which approaches the problem of dealing with presentation markup differently than EJS and Hogan.

10.4 Templating with Jade

Jade⁵ offers an alternative way to specify HTML. The key difference between Jade and the majority of other templating systems is the use of meaningful whitespace. When creating a template in Jade, you use indentation to indicate HTML tag nesting. Indentation tags don't have to be explicitly closed, which eliminates the problem of accidentally closing tags prematurely, or not at all. Using indentation also results in templates that are less visually dense and easier to maintain.

Footnote 5 <http://jade-lang.com>

For a quick example of this at work, let's look at how you'd represent the following snippet of HTML:

```
<div id="main" class="content bob"><strong>"Hello world!"</strong></div>
```

This HTML could be represented using the following Jade template:

```
div.content#main
  strong "Hello world!"
```

Jade, like EJS, allows you to embed JavaScript, and you can use it on the server- or client-side. But Jade offers additional features such as support for template inheritance and “mixins.” Mixins allow you to define easily reusable mini-templates to represent the HTML used for commonly occurring visual elements, such as item lists and boxes. Mixins are very similar in concept to the Hogan.js “partials” which you learned about in the previous section. Template inheritance makes it easy to organize the Jade templates needed to render a single HTML page into multiple files. You'll learn about these features in detail later in this section.

To install Jade in a Node application directory, enter the following into the command line:

```
npm install jade
```

Installing Jade with the `-g` global flag is useful as well as it gives you access to a `jade` command-line tool that will allow you to quickly render a template to HTML. The following command-line use would result in the `template/sidebar.jade` file being rendered to `sidebar.html` in the `template` directory. The Jade command-line tool gives you an easy way to experiment with Jade syntax.

```
jade template/sidebar.jade
```

In this section you'll learn:

- Jade basics such as specifying class names, attributes, and block expansion.
- How to add logic to your Jade templates using built-in keywords.
- How to organize your templates using inheritance, blocks and mixins.

To get started, let's look at the basics of Jade usage and syntax.

10.4.1 Jade basics

Jade uses the same tag names as HTML, but lets you lose the opening and closing `<` and `>` characters and uses indentation to express tag nesting.

A tag can have one or more CSS classes associated with it by adding `.<classname>.` A `div` element with the “content” and “sidebar” classes applied to it would be represented using the following:

```
div.content.sidebar
```

CSS IDs are assigned by adding “`#<ID>`” to the tag. You'd add a CSS ID of “`featured_content`” to the previous example using the following Jade

representation.

```
div.content.sidebar#featured_content
```

SIDE BAR

DIV tag shorthand

Because the `div` tag is commonly used in HTML, Jade offers a shorthand way of specifying it. The following example will render to the same HTML as the previous example:

```
.content.sidebar#featured_content
```

Now that you know how to specify HTML tags and their CSS classes and IDs, let's look at how to specify HTML tag attributes.

SPECIFYING TAG ATTRIBUTES

Specifying tag attributes is done by enclosing the attributes in parentheses, separating the specification of each attribute with a comma. You can specify a hyperlink that'll open in a different tab by using the following Jade representation:

```
a(href='http://nodejs.org', target='_blank')
```

As the specification of tag attributes can lead to long lines of Jade, the template engine provides you with some flexibility. The following Jade is valid and equivalent to the previous example:

```
a(href='http://nodejs.org',
target='_blank')
```

You can also specify attributes that don't require a value. The following Jade example shows the specification of an HTML form that includes a `select` element with an option preselected:

```
strong Select your favorite food:  
form  
  select  
    option(value='Cheese') Cheese  
    option(value='Tofu', selected) Tofu
```

SPECIFYING TAG CONTENT

In the previous code snippet, we also showed examples of tag content: “Select your favorite food:” after the `strong` tag; “Cheese” after the first `option` tag; and the tag content “Tofu” after the second `option` tag.

This is the normal way to specify tag content in Jade, but not the only way. Although this style is great for short bits of content, it can result in Jade templates with overly long lines if a tag’s content is lengthy. Luckily, as the following example shows, Jade will allow you to specify tag content using the `|` character:

```
textarea  
| This is some default text  
| that the user should be  
| provided with.
```

If the HTML tag, like the `style` and `script` tags, only ever accepts text (i.e. does not allow nested HTML elements) then the `|` characters can be left out entirely as the following example shows:

```
style  
  h1 {  
    font-size: 6em;  
    color: #9dff0c;  
  }
```

Having two separate ways to express long tag content and short tag content helps you keep your Jade templates elegant. Jade also supports an alternate way to express nesting called “block expansion.”

KEEPING IT ORGANIZED WITH "BLOCK EXPANSION"

Jade normally expresses nesting through indentation but sometimes indentation can lead to excess whitespace.

For example, here is a Jade template which uses indentation to define a simple list of links:

```
ul
  li
    a(href='http://nodejs.org/') Node.js homepage
  li
    a(href='http://npmjs.org/') NPM homepage
  li
    a(href='http://nodebits.org/') Nodebits blog
```

A more compact way to express the previous example is by using a Jade feature called “block expansion.” With block expansion, you add a colon after your tag to indicate nesting. The following code generates the same output as the previous listing in four lines instead of seven:

```
ul
  li: a(href='http://nodejs.org/') Node.js homepage
  li: a(href='http://npmjs.org/') NPM homepage
  li: a(href='http://nodebits.org/') Nodebits blog
```

Now that you’ve had a good look at how to represent markup using Jade, let’s look at how to integrate Jade with your web application.

INCORPORATING DATA IN JADE TEMPLATES

Data is relayed to the Jade engine in the same basic way as EJS. The template is first compiled into a function that is then called with a context in order to render the HTML output. The following is an example of this:

```
var jade = require('jade');
var template = 'strong #{message}';
var context = {message: 'Hello template!'};
```

```
var fn = jade.compile(template);
console.log(fn(context));
```

In the previous example, the `#{{message}}` in the template specified a placeholder to be replaced by a context value.

Context values can also be used to supply values for attributes. The following example would render ``.

```
var jade = require('jade');
var template = 'a(href = url)';
var context = {url: 'http://google.com'};

var fn = jade.compile(template);
console.log(fn(context));
```

Now that you've learned how HTML is represented using Jade and how you can provide Jade templates with application data, let's look at how you can incorporate logic in Jade.

10.4.2 Logic in Jade templates

Once you supply Jade templates with application data, you need logic to deal with that data. Jade allows you to directly embed lines of JavaScript code into your templates which is how you define logic in your templates. Code like `if` statements, `for` loops and `var` declarations are common.

Let's first look at the different ways Jade handles output when embedding JavaScript code.

USING JAVASCRIPT IN JADE TEMPLATES

Prefixing a line of JavaScript logic with `-` will execute the JavaScript without including any value returned from the code in the template's output. Prefixing JavaScript logic with `=` will include a value returned from code, escaped to prevent XSS attacks. But if your JavaScript generates code that shouldn't be escaped, you can prefix it with `!=`. Table 10.1 summarizes these prefixes:

Table 10.1 Prefixes used to embed JavaScript in Jade

Prefix	Output
=	Escaped output (for untrusted or unpredictable values, XSS safe)
!=	Output without escaping (for trusted or predictable values)
-	No output

Jade includes a number of commonly used conditional and iterative statements that can be written without prefixes. They are:

- if
- else if
- else
- case
- when
- default
- until
- while
- each
- unless

Jade also allows you to define variables. The following shows two ways of assignment that are equivalent in Jade:

```
- var count = 0
count = 0
```

The unprefixed statements have no output just like the – prefix above.

ITERATING THROUGH OBJECTS AND ARRAYS

Values passed in a context are accessible to JavaScript in Jade. In the following example we'll read a Jade template from a file and pass the Jade template a context containing a couple of messages that we intend to display in an Array:

```

var jade = require('jade');
var fs = require('fs');
var template = fs.readFileSync('./template.jade');
var context = { messages: [
  'You have logged in successfully.',
  'Welcome back!'
]};
var fn = jade.compile(template);
console.log(fn(context));

```

The Jade template would contain the following:

```

- messages.forEach(function(message) {
  p= message
- })

```

The final HTML output would be:

```
<p>You have logged in successfully.</p><p>Welcome back!</p>
```

Jade also supports a non-JavaScript form of iteration: the `each` statement. `each` statements allow you to cycle through arrays and object properties with ease.

The following is equivalent to the previous example but using `each` instead:

```

each message in messages
  p= message

```

You can perform cycling through object properties using a slight variation, as the following example shows:

```

each value, key in post
  div

```

```
strong #{key}
p value
```

CONDITIONALLY RENDERING TEMPLATE CODE

Sometimes templates need to make decisions on how data is displayed depending on the value of data. The following example illustrates a conditional in which, roughly 50 percent of the time, the script tag is outputted as HTML:

```
- var n = Math.round(Math.random() * 1) + 1
- if (n == 1) {
  script
    alert('You win!');
- }
```

Conditionals can also be written in Jade using the following cleaner, alternate form:

```
- var n = Math.round(Math.random() * 1) + 1
if n == 1
  script
    alert('You win!');
```

If you're writing a negated conditional, for example `if (n != 1)`, you could use Jade's `unless` keyword:

```
- var n = Math.round(Math.random() * 1) + 1
unless n == 1
  script
    alert('You win!');
```

USING CASE STATEMENTS IN JADE

Jade also supports a non-JavaScript form of conditional similar to a `switch`: the `case` statement. `case` statements allow you specify an outcome based on a number of template scenarios.

The following example template shows how the case statement can be used to display results from the search of a blog in three different ways. If the search finds nothing, a message is shown indicating that. If a single blog post is found, it's displayed in detail. If multiple blog posts are found, an each statement is used to iterate through the posts, displaying their titles.

```
case results.length
when 0
  p No results found.
when 1
  p= results[0].content
default
  each result in results
    p= result.title
```

10.4.3 Organizing Jade templates

With your templates defined, you next need to know how to organize them. As with application logic, you don't want to make your template files overly large. A single template file should correspond to a conceptual building block: a page, a sidebar, or blog post content for example.

In this subsection you'll learn a few mechanisms that allow different template files to work together to render content:

- Structuring multiple templates with template inheritance
- Implementing layouts using block prepending/appending
- Template including
- Reusing template logic with mixins

STRUCTURING MULTIPLE TEMPLATES WITH TEMPLATE INHERITANCE

Template inheritance is one means of structuring multiple templates. The concept treats templates, conceptually, like classes in the object-oriented programming paradigm. One template can extend another, which can in turn extend another. You can use as many levels of inheritance as makes sense.

As a simple example, let's look at using template inheritance to provide a basic HTML wrapper with which to wrap page content. In a working directory, create a folder called `template` in which you'll put the example's Jade file. For a page template you'll create a file called `layout.jade` containing the following Jade:

```

html
  head
    block title
  body
    block content

```

The `layout.jade` template contains the bare-bones definition of an HTML page as well as two “blocks.” Blocks are used in template inheritance to define where a descendant template can provide content. In `layout.jade` there is a “title” block, allowing a descendant template to set the title, and a “content” block, allowing a descendant template to set what is to be displayed on the page.

Next, in your working directory’s `template` directory, create a file named `page.jade`. This template file will populate the “title” and “content” blocks:

```

extends layout

block title
  title Messages

block content
  each message in messages
    p= message

```

Finally, add the logic in listing 10.12 (a modification of an earlier example in this section), which will display the template results, showing inheritance in action:

Listing 10.12 inheritance.js: Template inheritance in action

```

var jade = require('jade');
var fs = require('fs');
var templateFile = './template/page.jade';
var iterTemplate = fs.readFileSync(templateFile);
var context = {messages: [
  'You have logged in successfully.',
  'Welcome back!'
]};

var iterFn = jade.compile(
  iterTemplate,
  {filename: templateFile}
);

```

```
console.log(iteratorFn(context));
```

Now that you've seen template inheritance in action, let's look at another template inheritance feature: block prepending and appending.

IMPLEMENTING LAYOUTS USING BLOCK PREPENDING/APPPENDING

In the previous example the blocks in `layout.jade` contained no content, which made setting the content in the `page.jade` template straightforward. But if a block in an inherited template does contain content, this content can be built upon, rather than replaced, by descendent templates using block prepending and appending. This allows you to define common content and add to it, rather than replace it.

The following `layout.jade` template contains an additional block, "scripts," which contains content: a `script` tag that'll load the jQuery JavaScript library:

```
html
  head
    block title
    block scripts
      script(src='//ajax.googleapis.com/ajax/libs/jquery/1.8/jquery.js')
  body
    block content
```

If you want the `page.jade` template to additionally load the jQuery UI library, you could do this by using the template in the following listing 10.13:

Listing 10.13 block_append.js: Using block append to load an additional JavaScript file

```
extends layout ①
baseUrl = "http://ajax.googleapis.com/ajax/libs/jqueryui/1.8/"

block title
  title Messages

block style ②
  link(rel="stylesheet", href= baseUrl+"themes/flick/jquery-ui.css")

block append scripts ③
```

```

script(src= baseUrl+"jquery-ui.js")

block content
count = 0
each message in messages
- count = count + 1
script
$(function() {
  $("#message_#{count}").dialog({
    height: 140,
    modal: true
  });
})
!= '<div id="message_' + count + '">' + message + '</div>'

```

- ① This template extends the layout template
- ② Define the style block
- ③ Append this scripts block to the one defined in layout

But template inheritance isn't the only way to integrate multiple templates. You also can use the `include` Jade command.

TEMPLATE INCLUDING

Another tool for organizing templates is Jade's `include` command. This command incorporates the contents of another template. If you added the line `include footer` to the `layout.jade` template from the earlier example you'd end up with the following template:

```

html
  head
    block title
    block style
    block scripts
      script(src='//ajax.googleapis.com/ajax/libs/jquery/1.8/jquery.js')
  body
    block content
    include footer

```

This template would include the contents of a template named `footer.jade` in the rendered output of `layout.jade`, as illustrated in figure 10.4:

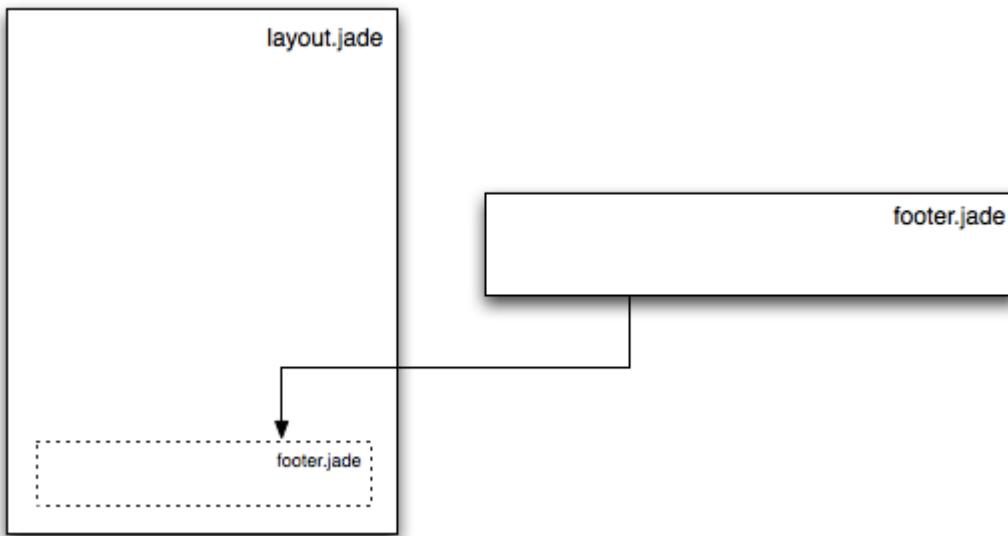


Figure 10.4 Jade's include mechanism provides a simple way to include the contents of one template in another template during rendering

This could be used, for example, to add information about the site, or design elements, to `layout.jade`. You can also include non-Jade files by specifying the file extension (e.g. `include twitter_widget.html`).

REUSING TEMPLATE LOGIC WITH MIXINS

Although Jade's `include` command is useful for bringing in previously created chunks of code, it's not ideal for creating a library of reusable functionality that you can share between pages and applications. For this, Jade provides the `mixin` command which lets you define reuseable Jade snippets.

A Jade mixin is analogous to a JavaScript function. A mixin can, like a function, take arguments. These arguments can be used to generate Jade code.

Let's say, for example, your application handles a data structure similar to the following:

```
var students = [
  {name: 'Rick LaRue', age: 23},
  {name: 'Sarah Cathands', age: 25},
  {name: 'Bob Dobbs', age: 37}
];
```

If you want to define a way to output an HTML list derived from a given property of each object you could define a mixin like the following one to

accomplish this:

```
mixin list_object_property(objects, property)
  ul
    each object in objects
      li= object[property]
```

You could then use the mixin to display the data using the following line of Jade:

```
mixin list_object_property(students, 'name')
```

By using template inheritance, `include` statements, and mixins you can easily reuse presentation markup and can prevent your template files from becoming larger than they need to be.

10.5 Summary

Now that you've learned how three popular HTML template engines work, you can use the technique of templating to keep your application logic and presentation organized. The Node community has created many template engines, which means if there's something you don't like about the three you've tried in this chapter you may want to check out other engines⁶.

Footnote 6 <https://npmjs.org/browse/keyword/template>

The Handlebars.js⁷ template engine, for example, extends the Mustache templating language, adding additional features such as conditional tags and globally available lambdas. Dustjs⁸ prioritizes performance and features such as streaming. For a list of Node template engines, check out the consolidate.js project⁹, which provides an API that abstracts the use of template engines, making it easy to use multiple engines in your applications. But if the idea of having to learn any kind of template language at all seems distasteful, an engine called Plates¹⁰ allows you to stick to HTML, using its engine's logic to map application data to CSS IDs and classes within your markup.

Footnote 7 <https://github.com/wycats/handlebars.js/>

Footnote 8 <https://github.com/akdubya/dustjs>

Footnote 9 <https://github.com/visionmedia/consolidate.js>

Footnote 10 <https://github.com/flatiron/plates>

If you find Jade's way of dealing with the separation of presentation and application logic appealing, you might also want to look at Stylus¹¹, a project that takes a similar approach to dealing with the creation of CSS.

Footnote 11 <https://github.com/LearnBoost/stylus>

You now have the final piece you need to create professional web applications. In the next chapter we'll look at deployment: how to make your application available to the rest of the world.

Deploying Node applications and maintaining uptime



This chapter covers:

- Choosing where to host your Node application
- Deploying a typical application
- Maintaining uptime and maximizing performance

Developing a web application is one thing, but putting it into production is another. Every web platform requires knowledge of tips and tricks that increase stability and maximize performance, and Node is no different.

When you're faced with deploying a web application you'll find yourself considering where to host it. You'll want to consider how to monitor your application and keep it running. You may also wonder what you can do to make it as fast as possible. In this chapter you'll learn how to address these concerns for your Node web application. You'll learn how to:

- get your Node application up and running in public,
- how to deal with unexpected application termination, and
- how to get respectable performance.

To start, let's look at where you might choose to host your application.

11.1 Hosting Node applications

Most web application developers are familiar with PHP-based applications. When an Apache server with PHP support gets an HTTP request it'll map the path portion of the requested URL to a specific file and PHP will execute the contents of the file. This functionality makes it easy to deploy PHP applications: you upload PHP files to a certain location of the filesystem and they become accessible via web browsers. In addition to being easy to deploy, PHP applications can also be hosted cheaply as servers are often shared between a number of users.

But currently, deploying Node applications isn't that simple or cheap. Whereas Apache is usually provided by hosting companies to handle HTTP, Node must handle HTTP itself. You either need to set up and maintain a Linux server or use Node-specific cloud hosting services offered by companies like Joyent, Heroku, Nodejitsu, VMware, and Microsoft. Node-specific cloud hosting services are worth looking into if you want to avoid the trouble of administering your own server or want to benefit from Node-specific diagnostics, such as Joyent SmartOS's ability to measure what logic in a Node application performs slowest. The Cloud9 website, itself built using Node.js, offers a browser-based integrated development environment (IDE) in which you can clone projects from GitHub, work on them via the browser, then deploy them to a number of Node-specific cloud hosting services, as shown in the following table 11.1:

Table 11.1 Node-specific cloud hosting and IDE services

Name	Website
Heroku	http://www.heroku.com/
Nodejitsu	http://nodejitsu.com/
VMware's Cloud Foundry	http://www.cloudfoundry.com/
Microsoft Azure SDK for Node.js	http://www.windowsazure.com/en-us/develop/nodejs/
Cloud9 IDE	http://c9.io/

Currently, running your own Linux server offers the most flexibility because you can easily install any related applications you need, such as database servers. Node-specific cloud hosting only offers you the choice of a small selection of related applications.

Linux server administration is its own realm of expertise and, if you choose to handle your own deployment, it's advisable to read up on your chosen Linux variant to be sure you're familiar with setup and maintenance procedures.

TIP**VirtualBox**

If you're new to server administration, you can experiment by running software like VirtualBox¹, which allows you to run a virtual Linux computer on your workstation, no matter what operating system your workstation runs.

Footnote 1 <https://www.virtualbox.org/>

If you're familiar with Linux server options, you may want to skim until you get to section 11.2, which is where we start to talk about cloud hosting. Right now though, let's talk about the options available to you:

- dedicated servers
- virtual private servers
- general-purpose cloud servers

11.1.1 Dedicated and virtual private servers

Your server may either be a physical one, commonly known as a “dedicated,” server, or a virtual one. Virtual servers run inside physical servers and are assigned a share of the physical server’s RAM, processing power, and disk space. Virtual servers emulate a physical server and you can administer them in the same way. More than one virtual server can run inside a physical server.

Dedicated servers are usually more expensive than virtual servers and often require more setup time as components may have to be ordered, assembled, and configured. Virtual private servers (VPSs), on the other hand, can be set up quickly as they are created inside preexisting physical servers.

VPSs are a good hosting solution for web applications if you don't anticipate a quick growth in usage. VPSs are cheap and can be easily allocated additional

resources, when needed, such as disk space and RAM. The technology is established and many companies, such as Linode² and Prgmr³ make it easy to get up and running.

Footnote 2 <http://www.linode.com/>

Footnote 3 <http://prgmr.com/xen/>

VPSs, like dedicated servers, can't usually be created on-demand. Being able to handle quick growth in usage requires the ability to quickly add more servers, without relying on human intervention. For the ability to handle this you'll need to use "cloud" hosting.

11.1.2 Cloud hosting

Cloud servers are similar to VPSs in that they are virtual emulations of dedicated servers. But they have an advantage over dedicated servers and VPSs in that their management can be fully automated. Cloud servers can be created, stopped, started, and destroyed using a remote interface or API.

Why would you need this? Let's say, for example, that you've founded a company that has created Node-based corporate intranet software. You'd like clients to be able to sign up for your service and, shortly after sign up, receive access to their own server running your software. You could hire technical staff to set up and deploy servers for these clients around the clock, but unless you maintained your own data center they'd still have to coordinate with dedicated or VPS server providers to provide the needed resources in a timely manner. By using cloud servers you could have a management server send instructions via an API to your cloud hosting provider to give you access to new servers as needed. This level of automation enables you to deliver service to the customer quickly and without human intervention. Figure 11.1 visually represents how you can use cloud hosting to automate creating and destroying an application's servers.

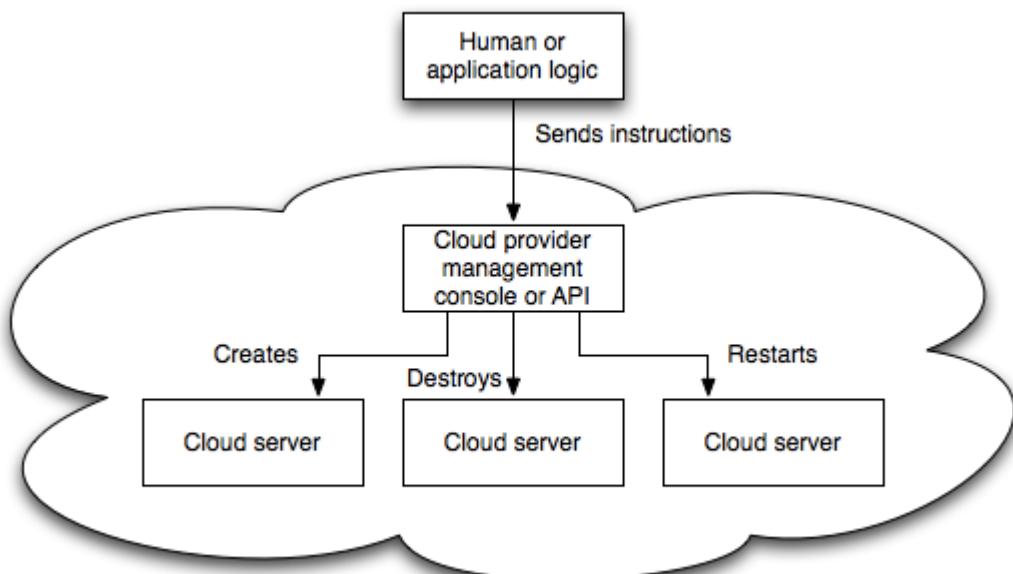


Figure 11.1 Creating, starting, stopping, and destroying cloud servers can be fully automated

The downside to using cloud servers is they tend to be more expensive than VPSs and can require some knowledge specific to the cloud platform.

AMAZON WEB SERVICES

The oldest and most popular cloud platform is Amazon Web Services (AWS)⁴. AWS itself consists of a range of different hosting-related services, like email delivery, content-delivery networks and lots more. Amazon's Elastic Cloud Computing (EC2), one of AWS's central services, allows you to create servers in the cloud whenever you need them.

Footnote 4 <http://aws.amazon.com/>

EC2 virtual servers are called “instances” and can be managed using either the command line or a web-based control console, shown in figure 11.2. As command-line use of AWS takes some time to get used to, the web-based console is recommended for first time users.

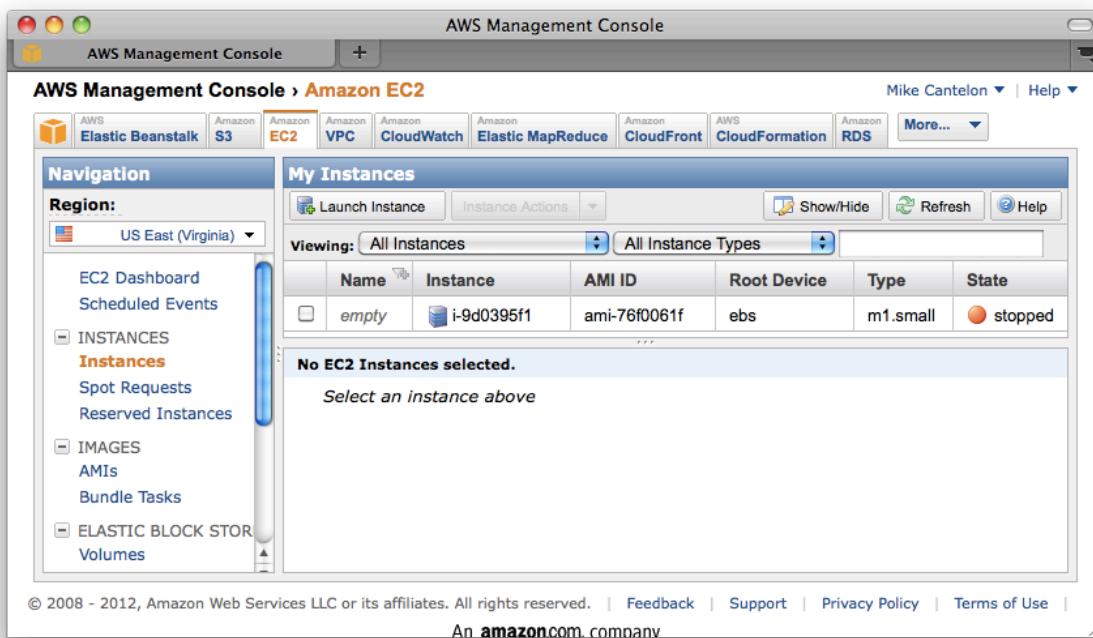


Figure 11.2 The Amazon Web Services web console provides an easier way to manage Amazon cloud servers for new AWS users than the command line

Luckily, because AWS's are ubiquitous, it's easy to get help online and find related tutorials such as Amazon's "Get Started with EC2"⁵.

Footnote 5 <http://docs.amazonwebservices.com/AWSEC2/latest/GettingStartedGuide/Welcome.html>

RACKSPACE CLOUD

A more basic, easier-to-use cloud platform is Rackspace Cloud⁶. Although a gentler learning curve may be appealing, Rackspace Cloud offers a smaller range of cloud-related products and functionality than does AWS and has a somewhat clunkier web interface. Rackspace Cloud servers either can be managed using a web interface, or with community-created command-line tools. Table 11.2 summarizes the hosting options:

Footnote 6 <http://www.rackspace.com/cloud/>

Table 11.2 Summary of hosting options

Suitable traffic growth	Hosting option	Cost
Slow	Dedicated	\$\$
Linear	Virtual private server	\$
Unpredictable	Cloud	\$\$\$

Now that you've had an overview of where you can host your Node application, let's look at exactly how you'd get your Node application running on a server.

11.2 Deployment basics

Suppose you've created a web application that you want to show off or maybe you've created a commercial application and need to test it before putting it into full production. You'll likely start with a simple deployment, and then do some work later to maximize uptime and performance. In this section, we'll walk you through a simple, temporary Git deployment, as well as keeping the application up and running with Forever. Temporary deploys don't persist beyond reboots, but they have the advantage of being quick to set up.

11.2.1 Deploying from a Git repository

Let's quickly go through a basic deployment using a Git repository to give you a feel for the fundamental steps.

Deployment is most commonly done by:

- Connecting to a server using SSH
- Installing Node and version control tools, like Git or Subversion, on the server (if needed)
- Downloading application files, including Node scripts and static assets like images and CSS stylesheets, from a version control repository to the server
- Starting the application

To follow is an example of an application start after downloading the application files using Git:

```
git clone https://github.com/Marak/hellonode.git
cd hellonode
node server.js
```

Like PHP, Node doesn't run as a background task. Because of this, the basic deployment we outlined would require keeping the SSH connection open. As soon as the SSH connection closes, the application will terminate. Luckily, it's fairly easy to keep your application running using a simple tool.

NOTE**Automating deployment**

If you'd like to automate deployment of your Node application you can take a number of approaches. One way is to use a tool like Fleet⁷, which allows you to deploy, using `git push`, to one or more servers. A more traditional way is by using Capistrano, as detailed in this blog post⁸.

Footnote 7 <https://github.com/substack/fleet>

Footnote 8

<http://blog.evantahler.com/deploying-node-js-applications-with-capistrano>

11.2.2 Keeping Node running

Let's say you've created a personal blog, using the Cloud 9 "Nog" blogging application⁹, and want to deploy it, making sure that it stays running even if you disconnect from SSH or it crashes.

Footnote 9 <https://github.com/c9/nog>

The most popular tool in the Node community for dealing with this is Nodejitsu's Forever¹⁰. It keeps your application running after you disconnect from SSH and, additionally, restarts it if it crashes. Figure 11.3 shows, conceptually, how Forever works.

Footnote 10 <https://github.com/nodejitsu/forever>

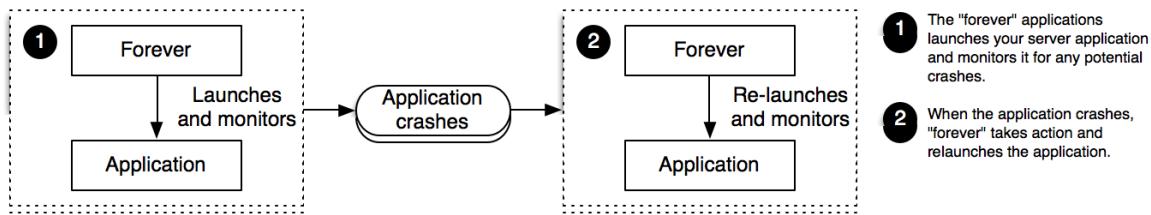


Figure 11.3 The Forever tool helps you keep your application running, even if it crashes

You can install Forever globally using the `sudo` command.

NOTE

The "sudo" command

Often when installing an `npm` module *globally* (with the `-g` flag), you will need to prefix the command with the `sudo`¹¹ command in order to run the command with superuser privileges. The first time you use the `sudo` command you will be prompted to enter your password, and after that then the command specified after it will be run.

Footnote 11 <http://www.sudo.ws/>

If you're following along, install it now using the following command:

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

Once you've installed Forever, you could use it to start your blog, and keep it running, using the following command:

```
forever start server.js
```

If you wanted to stop your blog for some reason, you'd then use Forever's stop command:

```
forever stop server.js
```

When using Forever, you can get a list of what applications the tool is managing by using its “list” command:

```
forever list
```

Another useful capability of Forever is the optional ability to have your application restarted when any source files have changed. This frees you from having to manually restart each time you add a feature or fix a bug.

To start Forever in this mode, use the `-w` flag:

```
forever -w start server.js
```

Although Forever is an extremely useful tool for deploying applications, you may want to use something more fully featured for long-term deploys. In the next section we’ll look at more “industrial strength” monitoring solutions and how to maximize application performance.

11.3 Maximizing uptime and performance

Once a Node application is release-worthy, you’ll want to make sure it starts and stops when the server starts and stops as well as automatically restarts when the server crashes. It’s easy to forget to stop an application before a reboot or to forget to start an application after a reboot.

You’ll also want to make sure you’re taking steps to maximize performance. Modern servers, for example, are generally multi-core and it makes sense when you’re running your application on a server with a quad-core CPU that you’re not using only a single core. If you’re using only a single core and your web application’s traffic increases to the extent that the single core doesn’t have spare processing capability to handle the traffic, your web application won’t be able to consistently respond.

In addition to using all CPU cores, you’ll want to avoid using Node to host static files for high-volume production sites. Node can’t serve static files as efficiently as software optimized to do only this; instead, you’ll want to use

technologies like Nginx¹², which specializes in serving static files. You could also simply upload all your static files to a Content Delivery Network (CDN)¹³, like Amazon S3, and reference those files in your application.

Footnote 12 <http://nginx.org/en/>

Footnote 13 http://en.wikipedia.org/wiki/Content_delivery_network

In this section, we'll cover some server uptime and performance tips:

- Using Upstart to keep your application up and running through restarts and crashes
- Using Node's "cluster" API to utilize multi-core processors
- Serving Node application static files using NGINX

11.3.1 Maintaining uptime

Let's say you're happy with an application and want to market it to the world. You want to make dead sure that if you restart a server you don't then forget to restart your application. You also want to make sure that if your application crashes it's not only automatically restarted, but the crash is logged and you're notified, which allows you to diagnose any underlying issues.

In Ubuntu and CentOS, the two most popular Linux distributions for serving web applications, services are provided for application management and monitoring to help keep a tight rein on your applications. These services are mature, robust, and well-supported, and a proper deployment using them will maximize your uptime.

Let's start by looking at Upstart: Ubuntu's solution to maintaining uptime.

USING UPSTART TO START AND STOP YOUR APPLICATION

Upstart is a project that provides an elegant way to manage the starting and stopping of any Linux application, including Node applications. Modern versions of Ubuntu and CentOS 6 support the use of Upstart.

You can install Upstart on Ubuntu, if it's not already installed, using the following command:

```
sudo apt-get install upstart
```

You can install Upstart on CentOS, if it's not already installed, using the following command:

```
sudo yum install upstart
```

Once you've installed Upstart you'll need to add an Upstart configuration file for each of your applications. These files are created in the `/etc/init` directory and are named something like `my_application_name.conf`. The configuration files do not need to be marked as executable.

The following steps will create an empty Upstart configuration file for the previous example application:

```
sudo touch /etc/init/hellonode.conf
```

Now, add the contents of listing 11.1 to your config file. This setup specifies to run the application upon startup of the server, and stop it upon shutdown. The `exec` section is what gets executed by Upstart:

Listing 11.1 /etc/init/hellonode.conf: A typical Upstart configuration file

```
#!upstart
author      "mcantelon"    ①
description "hellonode"    ②
setuid      "nonrootuser"   ③

start on (local/filesystems and net-device-up IFACE=eth0) ④
stop on shutdown ⑤

respawn ⑥
console log ⑦
env NODE_ENV=production ⑧

exec /usr/bin/node /path/to/server.js ⑨
```

①

- 1 Application author name
- 2 Application name and/or description
- 3 Run the application using as user "nonrootuser"
- 4 Start application on startup after the file system and network is available
- 5 Stop application on shutdown
- 6 Restart application when it crashes
- 7 Log STDIN and STDERR to /var/log/upstart/yourapp.log
- 8 Set any environmental variables necessary to the application
- 9 Command to execute your application

This configuration will keep your process up and running, during server restarts and even if it crashes unexpectedly. All the application generated output will be sent to /var/log/upstart/hellonode.log and Upstart will manage the log rotation for you.

Now that you've created an Upstart configuration file you can start your application using the following command:

```
sudo start hellonode
```

If your application was started successfully you will see:

```
hellonode start/running, process 6770
```

Upstart is highly configurable. Checkout the online cookbook¹⁴ for all the available options.

Footnote 14 <http://upstart.ubuntu.com/cookbook/>

SIDE BAR**Upstart and respawning**

Upstart, by default when using the `respawn` option, will continually reload your application on crashes *unless the application is restarted 10 times within 5 seconds*. You can change this limit using the `respawn limit COUNT INTERVAL` option where `COUNT` is the number of times within the `INTERVAL`, which is in seconds. For example, 20 times in 5 seconds would be:

```
respawn
respawn limit 20 5
```

If your application is reloaded 10 times within 5 seconds (the default), typically there is something wrong in the code or configuration, and it will never start successfully. Upstart will not try to restart after reaching the limit in order to save resources for other processes. It is a good idea to do health checks outside of Upstart that ideally provide alerts through email or some other means of quick communication back to the development team. A health check, for a web application, can simply be hitting the website and seeing if you get a valid response. You could roll your own or use tools such as Monit¹⁵ or Zabbix¹⁶ for this.

Footnote 15 <http://mmonit.com/monit/>

Footnote 16 <http://www.zabbix.com/>

With the ability to keep your application running regardless of crashes and server reboots, the next logical concern is performance, which Node's cluster API can help with.

11.3.2 The cluster API: Taking advantage of multiple cores

Most modern computer CPUs have multiple cores. But a Node process uses only one of them when running. If you were hosting a Node application on a server and wanted to maximize the server's usage you could manually start multiple instances of your application on different TCP/IP ports, and use a load balancer to distribute web traffic to these different instances, but that's laborious to set up.

To make it easier to use multiple cores for a single application, the cluster API was added. This API makes it easy for your application to run multiple "workers" that each do the same thing, and respond to the same TCP/IP port, but can be run

simultaneously using multiple cores. Figure 11.4 visually represents how an application's processing would be organized, using the cluster API on a four-core processor.

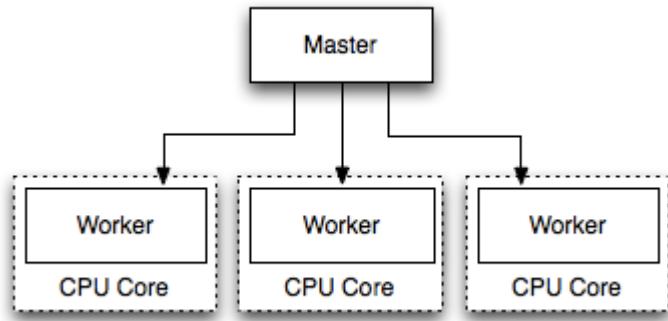


Figure 11.4 A visualization of a master spawning three workers on a four-core processor

Listing 11.2 automatically spawns a master process and a worker for each additional core.

Listing 11.2 cluster_demo.js: A demonstration of Node's cluster API

```

var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length; ①

if (cluster.isMaster) {
  for (var i = 0; i < numCPUs; i++) { ②
    cluster.fork();
  }

  cluster.on('death', function(worker) {
    console.log('Worker ' + worker.pid + ' died.');
  });
} else {
  http.Server(function(req, res) { ③
    res.writeHead(200);
    res.end('I am worker ID ' + process.env.NODE_CLUSTER_ID);
  }).listen(8000);
}
  
```

- ① Determine number of cores server has
- ② Create a fork for each core
- ③ Define work to be done by each worker

Because masters and workers run in separate operating system processes, which is necessary if they're to run on separate cores, they can't share state through global variables. But the cluster API does provide a means of communicating between the master and the workers.

Listing 11.3 shows an example use of the cluster API where messages are passed between the master and the workers. A count of all requests is kept by the master and whenever a worker reports handling a request, it is relayed to each worker.

Listing 11.3 cluster.messaging.js: A demonstration of Node's cluster API

```

var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;
var workers = {};
var requests = 0;

if (cluster.isMaster) {
  for (var i = 0; i < numCPUs; i++) {
    workers[i] = cluster.fork();

    (function (i) {
      workers[i].on('message', function(message) { ①
        if (message.cmd == 'incrementRequestTotal') {
          requests++; ②
          for (var j = 0; j < numCPUs; j++) {
            workers[j].send({ ③
              cmd:      'updateOfRequestTotal',
              requests: requests
            });
          }
        }
      });
    })(i); ④
  }

  cluster.on('death', function(worker) {
    console.log('Worker ' + worker.pid + ' died.');
  });
} else {
  process.on('message', function(message) { ⑤
    if (message.cmd == 'updateOfRequestTotal') {
      requests = message.requests; ⑥
    }
  });
}

http.Server(function(req, res) {

```

```

res.writeHead(200);
res.end('Worker ID ' + process.env.NODE_WORKER_ID
  + ' says cluster has responded to ' + requests
  + ' requests.');
process.send({cmd: 'incrementRequestTotal'}); ⑦
}).listen(8000);
}

```

- ① Listen for messages from the worker
- ② Increase request total
- ③ Send new request total to each worker
- ④ Use a closure to preserve the value of worker
- ⑤ Listen for messages from the master
- ⑥ Update request count using master's message
- ⑦ Let master know request total should increase

Node's cluster API is a simple way of creating applications that take advantage of modern hardware.

11.3.3 Hosting static files and proxying

Although Node is an effective solution for serving dynamic web content, it's not the most efficient way to serve static files such as images, CSS stylesheets, or client-side JavaScript. Serving static files over HTTP is a specific task for which specific software projects are optimized because they've focused primarily on this task for many years.

Fortunately nginx¹⁷, an open source web server optimized for serving static files, is easy to set up alongside Node to serve those files. In a typical nginx/Node configuration, nginx initially handles each web request, relaying requests that aren't for static files back to Node. This configuration is represented visually in figure 11.5:

Footnote 17 <http://nginx.org/>

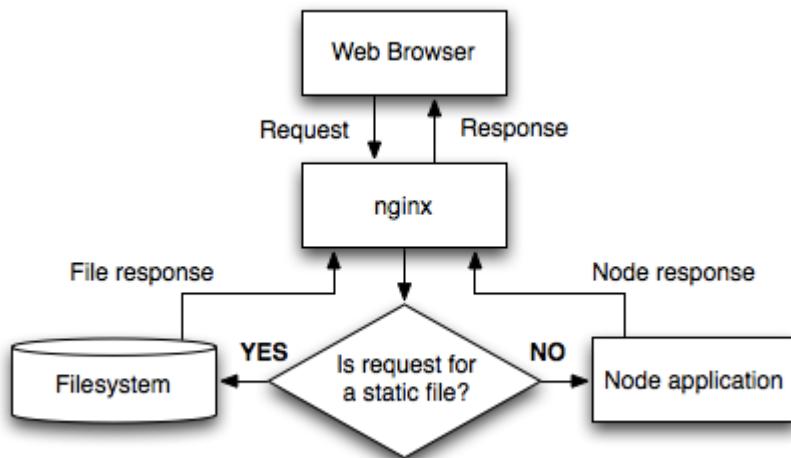


Figure 11.5 You can use nginx as a proxy to relay static assets quickly back to web clients

The configuration in the following listing 11.4, which would be put in the nginx configuration file's `http` section, implements this setup. The configuration file conventionally stores in a Linux server's `/etc` directory at `/etc/nginx/nginx.conf`.

Listing 11.4 `nginx_proxy.conf`: An example configuration file that uses nginx to proxy Node.js and serve static files

```

http {
    upstream my_node_app {
        server 127.0.0.1:8000; ①
    }

    server {
        listen 80; ②
        server_name localhost domain.com;
        access_log /var/log/nginx/my_node_app.log;

        location ~ /static/ { ③
            root /home/node/my_node_app;
            if (!-f $request_filename) {
                return 404;
            }
        }

        location / { ④
            proxy_pass http://my_node_app;
            proxy_redirect off;
        }
    }
}

```

```

    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_set_header X-NginX-Proxy true;
}
}
}

```

- ① IP and port of Node application
- ② Port on which proxy will receive requests
- ③ Handle file requests for URL paths starting with “/static/”
- ④ Define URL path proxy will respond to

By using nginx to handle your static web assets, you’ll ensure that Node is dedicated to doing what it’s best at.

11.4 Summary

In this chapter, we’ve introduced you to a number of Node hosting options, including Node-specific hosting, dedicated, virtual private server hosting, and cloud hosting. Each option suits different use cases.

Once you’re ready to deploy a Node application to a limited audience, you can get up and running quickly, without having to worry about your application crashing, by using the Forever tool to supervise your application. But for long-term deployment you may want to automate your application’s starts and stops using Upstart.

To get the most of your server resources, you can take advantage of Node’s cluster API to run application instances simultaneously on multiple cores. If your web application requires serving static assets such as images and PDF documents, you may also want to run the Nginx server and proxy your Node application through it.

Now that you have a good handle on the in’s and out’s of Node web applications, including deployment, it’s a good time to look at all the other things Node can do. In the next chapter we’ll look at Node’s other applications, everything from building command-line tools to “scraping” data from websites.

12

Beyond Web Servers

In this chapter:

- Socket.IO
- TCP/IP networking in-depth
- Tools for interacting with the operating system environment
- Developing command-line tools

Node's asynchronous nature makes it possible to do I/O intensive tasks that may otherwise be difficult or inefficient to do in a synchronous environment. We have mostly covered HTTP applications in this book but what about other kinds of applications? What else is Node useful for? The truth is that Node is tailored not only to HTTP, but to all kinds of general purpose I/O. In practice, this means you can build practically any type of application using Node, for example command-line programs, system administrative scripts, and realtime web applications.

In this chapter you will learn how Node is capable of creating web servers that go beyond a traditional web server by communicating in realtime. You will also learn about some of the other APIs that Node provides that you can use to create other kinds of applications, like TCP servers or command-line programs.

Specifically, we will cover:

- Socket.IO, which brings cross-browser realtime communication to web browsers
- TCP/IP networking in-depth, so you can implement all kinds of network applications with Node
- The APIs that Node provides to interact with the operating system directly
- Command line tools, including how to develop and work with them

12.1 Socket.IO

Socket.IO¹ is arguably the most well known module in the Node community. People who are interested in creating "realtime web applications", but have never heard of Node, usually hear about Socket.IO sooner or later, which then brings them to Node itself. Socket.IO allows you to write realtime web applications using a bi-directional communication channel between the server and client. At its simplest, Socket.IO has an API very similar to the WebSocket API²³, but has built-in fallbacks for older browsers where such features did not yet exist. Socket.IO also provides convenient APIs for broadcasting, volatile messages, and a lot more. These features have made Socket.IO very popular for web-based browser games, chat type apps, and streaming applications.

Footnote 1 <http://socket.io>

Footnote 2 <http://www.websocket.org>

Footnote 3 <http://en.wikipedia.org/wiki/WebSocket>

HTTP is a stateless protocol, meaning that the client is only able to make single, short-lived requests to the server, and the server has no real notion of connected or disconnected users. This limitation has initiated the standardization of the WebSocket protocol, which specifies a way for browsers to maintain a full-duplex connection to the server, allowing both ends to send and receive data simultaneously. These APIs allow for a whole new breed of web applications utilizing realtime communication between the client and server.

The problem with the WebSocket protocol is it is not yet finalized, and while some browsers have begun shipping versions with WebSockets, there's still all the older versions out there, especially Internet Explorer. Socket.IO solves this by utilizing WebSockets when available in the browser, otherwise falling back to other browser-specific tricks to simulate the behavior that WebSockets provide, even in older browsers.

In this section, you will build up two sample applications using Socket.IO:

- A minimal Socket.IO application that pushes the server's time to connected clients.
- A Socket.IO application that triggers page refreshes when CSS files are edited.

After building the example apps, we'll show you a few more ways you can use Socket.IO by briefly revisiting the "upload progress" example from Chapter 4. Let's start with the basics.

12.1.1 Creating a minimal Socket.IO application

So let's say you wanted to build a quick little web application that constantly updated the browser in realtime with the server's UTC time. An app like this would be useful to identify a possible difference between the client's and server's clocks. Now try to think of how you could build this application using the `http` module or the frameworks you have learned about so far. While it is possible to get something working using a trick like long-polling, using Socket.IO provides a cleaner interface for accomplishing this. Implementing this app with Socket.IO is about as minimal as you can get, so let's built it.

You can install Socket.IO using `npm`:

```
npm install socket.io
```

Listing 12.1 shows the server-side code, so save this file for now and we can try it out when you have the client-side code as well.

Listing 12.1 clock-server.js: Socket.IO Server that updates its clients with the time

```
var app = require('http').createServer(handler);
var io = require('socket.io').listen(app);      ①
var fs = require('fs');
var html = fs.readFileSync('index.html', 'utf8');

function handler (req, res) {                  ②
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('Content-Length', Buffer.byteLength(html, 'utf8'));
  res.end(html);
}

function tick () {
  var now = new Date().toUTCString();          ③
  io.sockets.send(now);                      ④
}

setInterval(tick, 1000);                      ⑤

app.listen(8080);
```

① Upgrade the regular HTTP server into a Socket.IO server

②

- ▀ The HTTP server code always serves the 'index.html' file
- ③ Get the UTC representation of the current time
- ④ Send the time to all connected sockets
- ⑤ Run the tick function once per second

As you can see, Socket.IO minimizes the amount of extra code you need to add to the base HTTP server. It only took 2 lines of code involving the `io` variable (which is the variable for your Socket.IO server instance) to enable realtime messages between your server and clients. In this clock server example, you continually invoke the `tick()` function once per second to notify all the connected clients of the server's time.

The server code first reads the `index.html` file into memory, so we need to implement that now. Listing 12.2 shows the client-side of this application utilizing Socket.IO.

Listing 12.2 index.html: Socket.IO client that displays the server's broadcasted time

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="/socket.io/socket.io.js">
    </script>
    <script type="text/javascript">
      var socket = io.connect();          1
      socket.on('message', function (time) { 2
        document.getElementById('time').innerHTML = time;
      });
    </script>
  </head>
  <body>Current server time is: <b><span id="time"></span></b>
  </body>
</html>
```

- ① First connect to the Socket.IO server
- ② When a 'message' event is received, then the server has sent the time
- ③ Update the "time" span element with the server time

TRY IT OUT!

You are now ready to run the server. Fire it up with `node clock-server.js` and you'll see the text "info - socket.io started". This means that Socket.IO is set up and ready to receive connections, so open up your browser to the URL `http://localhost:8080/`. With any luck you will be greeted by something similar to figure 12.1. The time will be updated every second from the message received by the server. Go ahead and open another browser at the same time to the same URL and you will see the values change together in sync.

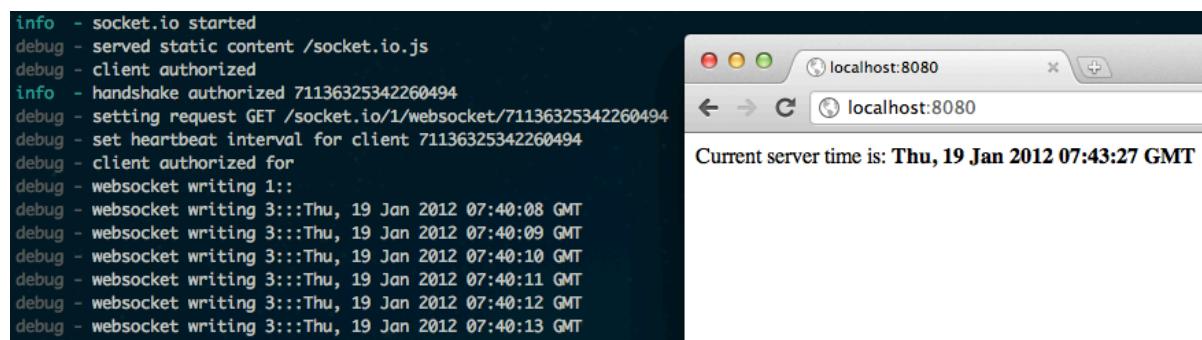


Figure 12.1 The clock server running in a Terminal with the browser connected to it

So just like that, realtime cross-browser communication between the client and server become possible in just a few lines of code, thanks to Socket.IO. Now that you have an idea of the simple things possible with Socket.IO, let's take a look at another example of how server-sent events are beneficial to developers.

NOTE

Other kinds of messaging with Socket.IO

Sending a message to all the connected sockets is only one way of interacting with the connected users that Socket.IO provides. There's also the ability to send messages to individual sockets, broadcast to all sockets except one, send volatile (optional) messages, and a lot more. Be sure to check out Socket.IO's documentation⁴ for more information.

Footnote 4 <http://socket.io/#how-to-use>

12.1.2 Using Socket.IO to trigger page and CSS reloads

Let's quickly take a look at the typical workflow for web page designers:

1. Open the web page in multiple browsers.
2. Look for styling on the page that needs adjusting.
3. Make changes to one or more stylesheets.

4. Manually reload *all* the web browsers.
5. Go back to step #2.

The biggest time waster in this list is definitely #4, where the designer needs to manually go into each web browser and click the "Refresh" button. This is especially time consuming when the designer needs to test different browsers on different computers, like a separate machine running Windows for testing Internet Explorer, and various mobile devices.

But what if you could eliminate this "manual refresh" step completely? Imagine that when you saved the stylesheet in your text editor, that *all* the web browsers that had that page opened automatically reloaded the changes in the CSS sheet. This would be a huge time saver for devs and designers alike, and Node's `fs.watchFile` and `fs.watch` functions make it possible in just a few lines of code.

Socket.IO matched with Node's `fs.watchFile()` function makes this possible with just a few lines of code. We will use `fs.watchFile()` in this example instead of the newer `fs.watch()` because we are assured this code will work the same on all platforms, but we'll cover that in depth later.

NOTE

`fs.watchFile()` vs. `fs.watch()`

Node.js provides two APIs for watching files: `fs.watchFile()`⁵ and `fs.watch()`⁶. `fs.watchFile` is rather expensive resource-wise, though more reliable and cross-platform. `fs.watch` is highly optimized for each platform, but has behavioral differences on certain platforms. We will go over these functions in greater detail in section 12.3.2.

Footnote 5

http://nodejs.org/api/fs.html#fs_fs_watchfile_filename_options_listener

Footnote 6

http://nodejs.org/api/fs.html#fs_fs_watch_filename_options_listener

In this example, we combine the Express framework with Socket.IO. Watch how they work together seamlessly in listing 12.3, just like the regular `http.Server` in the previous example. First let's take a look at the server code, in its entirety. Copy the file over and save it as `watch-server.js`, if you are interested in running this example at the end:

Listing 12.3 watch-server.js: Express and Socket.IO server that triggers "reload" and "stylesheet" events when files change

```

var fs = require('fs');
var url = require('url');
var http = require('http');
var path = require('path');
var express = require('express');
var app = express(); 1
var server = http.createServer(app);
var io = require('socket.io').listen(server); 2
var root = __dirname;

app.use(function (req, res, next) { 3
    req.on('static', function () { 4
        var file = url.parse(req.url).pathname;
        var mode = 'stylesheet';
        if (file[file.length - 1] == '/') {
            file += 'index.html';
            mode = 'reload';
        }
        createWatcher(file, mode); 5
    });
    next();
}); 6

app.use(express.static(root)); 6

var watchers = {};7

function createWatcher (file, event) {
    var absolute = path.join(root, file);

    if (watchers[absolute]) {
        return;
    }

    fs.watchFile(absolute, function (curr, prev) { 8
        if (curr.mtime !== prev.mtime) { 9
            io.sockets.emit(event, file);
        }
    });
}

watchers[absolute] = true; 10
}

server.listen(8080);

```

- ① First create the Express app server
- ② Then you wrap the HTTP server to create the socket.io instance
- ③ The app uses a little middleware to begin watching files that are returned by the static middleware
- ④ The "static" event gets emitted by the static() middleware, so register the event before it runs
- ⑤ This event handler determines the filename that was served and calls the createWatcher() function
- ⑥ The server is set up as a basic static file server
- ⑦ "watchers" is an Object that keeps a list of active files being watched
- ⑧ Begin watching the file, this callback gets invoked for any change to the file
- ⑨ Check to see if the 'mtime' (last modified time) changed before firing the Socket.IO event
- ⑩ Mark that this file is now being watched

So at this point you have a fully functional static file server, that is prepared to fire "reload" and "stylesheet" events using across the wire to the client using Socket.IO. Next take a look at the basic client-side code in listing 12.4. Save it as index.html so that it gets served at the root path, when you fire up the server next:

Listing 12.4 watch-index.html: The client-side code reloads the CSS stylesheets when events are received from the server

```
<!DOCTYPE html>
<html>
  <head>
    <title>Socket.IO dynamically reloading CSS stylesheets</title>
    <link rel="stylesheet" type="text/css" href="/header.css" />
    <link rel="stylesheet" type="text/css" href="/styles.css" />
    <script type="text/javascript" src="/socket.io/socket.io.js">
    </script>
    <script type="text/javascript">
      window.onload = function () {
        var socket = io.connect(); ①
        socket.on('reload', function () { ②
          window.location.reload();
        });
        socket.on('stylesheet', function (sheet) { ③
          var link = document.createElement('link');
          var head = document.getElementsByTagName('head')[0];
          link.setAttribute('rel', 'stylesheet');
          link.setAttribute('type', 'text/css');
          link.setAttribute('href', sheet);
        });
      }
    </script>
  </head>
  <body>
    <h1>Reloading CSS stylesheets</h1>
    <p>This page will automatically reload its CSS stylesheets whenever a file changes on the server. You can trigger a reload by clicking the button below or by changing a file in the static directory.</p>
    <button>Trigger Reload</button>
  </body>
</html>
```

```

        head.appendChild(link);
    });
}
</script>
</head>
<body>
<h1>This is our Awesome Webpage!</h1>
<div id="body">
<p>If this file (<code>index.html</code>) is edited, then the
server will send a message to the browser using Socket.IO telling
it to refresh the page.</p>

<p>If either of the stylesheets (<code>header.css</code> or
<code>styles.css</code>) are edited, then the server will send a
message to the browser using Socket.IO telling it to dynamically
reload the CSS, without refreshing the page.</p>
</div>
<div id="event-log"></div>
</body>
</html>

```

- ① Connect to the server
- ② The "reload" event is sent from the server
- ③ The "stylesheet" event is also sent from the server

TRYING IT OUT

Before this will work, you will need to create a couple of CSS files, `header.css` and `styles.css`, since the `index.html` file loads those two stylesheets when it loads. Any sorts of CSS will do, just make sure it affects the current layout of the page.

Now that you have the server code, the `index.html` file and the CSS stylesheets that the browser will use, you can try it out. Fire up the server:

```
$ node watch-server.js
```

Once the server has started, then open your web browser to `http://localhost:8080` and see simple HTML page being served and rendered. Now try altering one of the CSS files. Perhaps tweak a the background color of the `body` tag, and you will see the stylesheet reload in the browser right in front of your eyes, without even reloading the page itself. Try opening the page in multiple browsers at once!

In this example, "reload" and "stylesheet" are custom events that we have defined in the application; they are not specifically part of Socket.IO's API. This

demonstrates how the `socket` object acts like a bi-directional `EventEmitter`, which you can use to emit arbitrary events that `Socket.IO` will transfer across the wire for you. So now that you know this, it should be no surprise to hear that `socket.send()` simply emits a "message" event.

12.1.3 Other ideas for uses of `Socket.IO`

As you know, HTTP was never originally intended for any kinds of realtime communication. But with advances in browser technologies like WebSockets, and modules like `Socket.IO`, this limitation has been changed, opening a big door for all kinds of new applications that were never before possible in the web browser.

Back in Chapter 4 we said that using `Socket.IO` would be great for relaying upload progress events back to the browser for the user to see. You can apply the techniques used in the examples in this section to do this. Using a custom "progress" event would work well:

```
// ... updated from the example in 4.4.1
form.on('progress', function(bytesReceived, bytesExpected){
  var percent = Math.floor(bytesReceived / bytesExpected * 100);

  // here is where you can relay the uploaded percentage using Socket.IO
  socket.emit('progress', { percent: percent });
});
```

Getting access to the `socket` instance that matches the browser uploading the file is another task that will need to be figured out to make this progress relaying work, however it is outside the scope of this book. Do not fret though, there are resources on the internet⁷ that can aid in figuring that out.

Footnote 7 <http://www.danielbaulig.de/socket-ioexpress>

`Socket.IO` is game changing. As mentioned, developers interested in "realtime" web applications hear about `Socket.IO` before even knowing about `Node.js`, a testament to how influential and important it is. It is constantly gaining traction in web gaming communities and being used for more and more creative games and applications than one could ever have thought possible. It is also a very popular pick for use in applications written in `Node.js` competitions like `Node Knockout`⁸. What awesome *thing* will you write?

Footnote 8 <http://nodeknockout.com>

12.2 TCP/IP networking in-depth

Transitioning from Socket.io to the low-level networking APIs that Node provides out of the box may feel like a bit of a gear switch, and it is. But just like HTTP servers, Node is well suited for *any* kind of I/O bound task, and it is the same story for TCP-based networking. Node is a very good platform for writing, for example, an email server, file server, proxy server, just to name a few. But it can also be used as a client for these kinds of services as well. Node provides a few tools to aid in writing high quality and performant I/O applications, which you will learn about in this section.

Some networking protocols require reading values at the byte level like chars, ints, floats, etc., involving binary data. JavaScript falls short in this category by not including any native binary data type to work with. The closest you could get is crazy hacks with Strings. So Node picks up the slack by implementing its own `Buffer` data type, acting as a piece of fixed-length binary data, which makes it possible to access the low-level bytes needed to implement other protocols.

In this section you will learn about:

- Working with Buffers and binary data
- Creating a TCP server
- Creating a TCP client

12.2.1 Working with Buffers and binary data

"Buffers" are a special data type that Node provides for developers that act as a slab of raw binary data with a fixed length. They can be thought of as the equivalent of the `malloc()` C function⁹, or the `new` keyword in C++. Buffers are used all over the place throughout Node's core APIs, and are in fact returned in 'data' events by all "Stream" classes by default.

Footnote 9 http://wikipedia.org/wiki/C_dynamic_memory_allocation

Buffers are very fast and light objects. Node exposes the `Buffer` constructor globally, encouraging you to use it just like an extension of the regular JavaScript data types. From a programming point of view, you can think of them as similar to Arrays, except they are not resizable, and can only contain the numbers 0 through 255 as values. This makes them ideal for storing binary data of, well, anything really. Since buffers work with raw bytes, you can use them to implement any low-level protocol that you desire.

TEXT DATA VS. BINARY DATA

Say you wanted to store the number "121234869" into memory using a Buffer. By default Node assumes that you want to work with text-based data in Buffers, so when you pass the string "121234869" to the Buffer constructor function, a new buffer object will be allocated with the string value written to it.

```
var b = new Buffer("121234869");
console.log(b.length);
9
console.log(b);
<Buffer 31 32 31 32 33 34 38 36 39>
```

In this case, it would return a 9-byte buffer. This is because the string was written to the buffer using the default human-readable text-based encoding (UTF-8), where in this case, the string is represented with 1 byte per character.

Node also includes helper functions for reading and writing binary (machine-readable) integer data. These are needed for implementing machine-protocols that send raw data types (like ints, floats, doubles, etc.) over the wire. Since you want to store a number value in this case, it's possible to be more efficient by utilizing the helper function `writeInt32LE()` to write the number "121234869" as a machine-readable binary integer (assuming a little-endian processor) into a 4-byte Buffer. There are other variations of the Buffer helper functions as well, to name a few:

- `writeInt16LE()` for smaller integer values.
- `writeUInt32LE()` for unsigned values.
- `writeInt32BE()` for big-endian values.

There are lots more so be sure to check the Buffer API doc page¹⁰ if you're interested in them all.

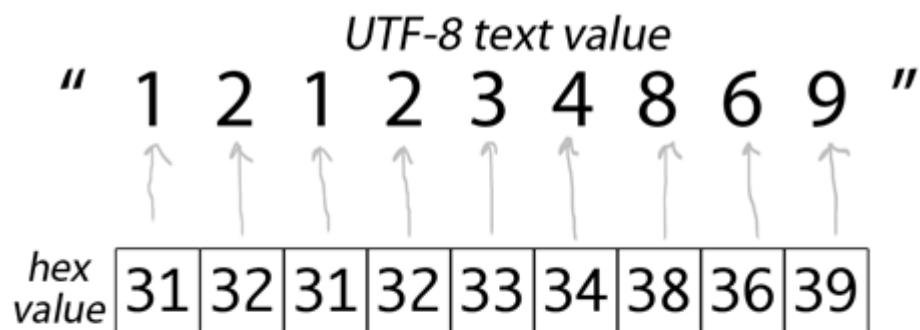
Footnote 10 <http://nodejs.org/docs/latest/api/buffer.html>

In this code snippet, the number will be written using the `writeInt32LE` binary helper function.

```
var b = new Buffer(4);
b.writeInt32LE(121234869, 0);
console.log(b.length);
4
```

```
console.log(b);
<Buffer b5 e5 39 07>
```

By storing the value as a binary integer instead of a text string in memory, it was possible to save over 50% of the memory used, from 9 bytes down to 4. Take a look at the breakdown of these two buffers in figure 12.2.



buffer with string data

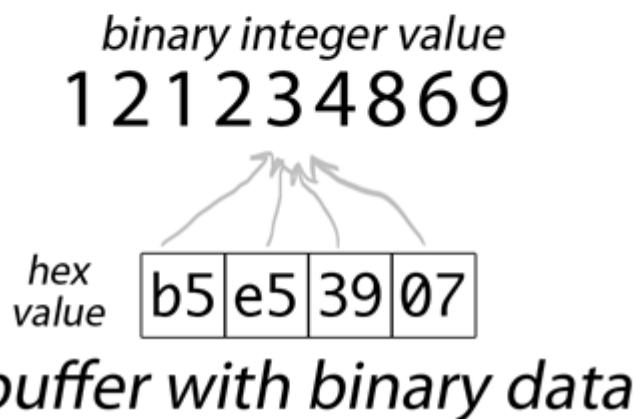


Figure 12.2 The difference between representing "121234869" as a text string vs. a little-endian binary integer at the byte-level

In essence, this is the difference between human-readable (text) protocols, and machine-readable (binary) protocols. But regardless of what kind of protocol you are working with, Node's `Buffer` class will be able to take care of handling the proper representation.

NOTE**Byte Endianness**

The term "endianness"¹¹ refers to the order of the bytes within a multibyte sequence. When bytes are ordered "little-endian" then that means the most significant byte is the last byte (right-to-left). Conversely, "big-endian" order is when the first byte is the most significant byte (left-to-right). Node.js offers equivalent helper functions for both little-endian and big-endian data types.

Footnote 11 <http://en.wikipedia.org/wiki/Endianness>

Now it's time to actually put these Buffer objects to use by creating a TCP server and interacting with it.

12.2.2 Creating a TCP server

Node's core API sticks to being low-level, with only the bare essentials exposed for modules to build on top of. Node's `http` module is a good example of this; building on top of the `net` module to implement the HTTP protocol. Other protocols, like SMTP for email or FTP for file transfer, will need to be implemented on top of the `net` module as well, since Node's core API does not implement any other higher level protocols.

WRITING DATA

The `net` module offers a raw TCP/IP socket interface for your applications to use. The API for creating a server is very similar to creating an HTTP server: you call `net.createServer()` and give it a callback function that will be invoked upon each connection. The main difference is that the callback function only takes one argument (usually named `socket`) which is the `Socket` object, as opposed to the `req` and `res` arguments when creating an HTTP Server.

NOTE**The Socket class**

The `Socket` class is used by both the client and server aspects of the `net` module in Node. It is a `Stream` subclass that is both `readable` and `writable` (bi-directional). That is, it emits "data" events when input data has been read from the socket, and has `write()` and `end()` functions for sending output data.

So first let's do something quick and silly, to show a "barebones" `net.Server` that waits for connections and then invokes our given callback

function. In this case, the application login inside the callback function simply writes "Hello World!" to the socket and closes the connection cleanly.

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

net.createServer(function (socket) {
  socket.write('Hello World!\r\n');
  socket.end();
}).listen(1337);
console.log('listening on port 1337');
```

So fire up the server for some testing:

```
$ node server.js
listening on port 1337
```

Now if you were to try to connect to the server in a web browser, it would not work since this server does not speak HTTP, only raw TCP. So in order to connect to this server and see our message you need to connect with a proper TCP client, like `netcat(1)`:

```
$ netcat localhost 1337
Hello World!
```

Great! Ok, now let's try using `telnet(1)`:

```
$ telnet localhost 1337
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^].
Hello World!
Connection closed by foreign host.
```

`telnet` is usually meant to be run in an interactive mode, so it prints out its own stuff as well, but the "Hello World" message does get printed right before the connection gets closed. Just as expected.

So writing data to the socket is easy, it's just `write()` calls and a final `end()` call. In case you didn't notice, this is exactly the same API as the HTTP `res` object when writing a response to the HTTP client.

READING DATA

It's common for servers to follow the "request -> response" paradigm, where the client connects and immediately sends a "request" of some sort. The server reads the request and processes a "response" of some sort to write back to the socket. This is exactly how the HTTP protocol works, as well as the majority of other networking protocols that are common in the wild, so it's important to know how to read data as well.

Fortunately, if you remember how to read a request body from an HTTP `req` object then reading from a TCP socket should be a piece of cake. Complying with the readable `Stream` interface, all you have to do is listen for "data" events which contain the input data that was read from the socket:

```
socket.on('data', function (data) {
  console.log('got "data"', data);
});
```

By default there is no encoding set on the `socket`, so the `data` argument will be a `Buffer` instance. Usually, this is exactly how you want it (that's why it's the default), but when it's more convenient you can call the `setEncoding()` function to have the `data` argument be the decoded Strings instead of `Buffers`. You also listen for the "end" event to know when the client has closed their end of the socket, and won't be sending any more data.

```
socket.on('end', function () {
  console.log('socket has ended');
});
```

We can easily write a quick TCP client that looks up the `version` string of the given SSH server, by simply waiting for the first "data" event.

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

var socket = net.connect({ host: process.argv[2], port: 22 });
socket.setEncoding('utf8');

socket.once('data', function (chunk) {
  console.log('SSH server version: %j', chunk.trim());
  socket.end();
```

```
} );
```

Now you can try it out. Note that this over-simplified example assumes that the entire version string will come in one chunk. Most of the time this works just fine, but a proper program would buffer the input until a char was found. Let's check what SSH server "github.com" uses:

```
$ node client.js github.com
SSH server version: "SSH-2.0-OpenSSH_5.5p1 Debian-6+squeezel+github8"
```

CONNECTING TWO STREAMS WITH SOCKET.PIPE()

Using `Stream#pipe()`¹² in conjunction with either the readable or writable portions of a `Socket` object is also a good idea. In fact, if you wanted to write a simple TCP server that simply echoed everything that was sent to it back to the client, then you could do that with a single line of code in your callback function:

Footnote 12 http://nodejs.org/api/stream.html#stream_stream_pipe_destination_options

```
socket.pipe(socket);
```

This example shows that it only takes one line of code to implement the IETF Echo Protocol¹³, but more importantly it demonstrates that you can `pipe()` both *to* and *from* the `socket` object. Though you would usually do so with more meaningful Stream instances, like a file-system or gzip stream. `Stream#pipe()` is one of the few times in Node's asynchronous APIs where doing the right thing ends up taking less code, so it is recommended to use it when possible!

Footnote 13 <http://tools.ietf.org/rfc/rfc862.txt>

HANDLING UNCLEAN DISCONNECTIONS

The last thing that should be said about TCP servers is anticipating clients that disconnect, but don't cleanly close the socket. In the case of `netcat(1)`, this would happen when you press Ctrl+C to kill the process, rather than pressing Ctrl+D to cleanly close the connection. To detect this situation you listen for the "close" event.

```
socket.on('close', function () {
  console.log('client disconnected');
});
```

If you have cleanup to do after a socket disconnects then you should do it from the "close" event, not the "end" event since it will not fire when the connection was not closed cleanly.

PUTTING IT ALL TOGETHER

Let's take all these events and create a simple echo server that logs stuff to the terminal when the various events occur, show in listing 12.5:

Listing 12.5 `tcp-echo-server.js`: A simple TCP server that echoes any data it receives back to the client

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

net.createServer(function (socket) {
  console.log('socket connected!');
  socket.on('data', function (data) {
    console.log('"data" event', data);
  });
  socket.on('end', function () {
    console.log('"end" event');
  });
  socket.on('close', function () {
    console.log('"close" event');
  });
  socket.on('error', function (e) {
    console.log('"error" event', e);
  });
  socket.pipe(socket);
}).listen(1337);
```

1 "data" can happen multiple times

2 "end" can only happen once per socket

3 "close" can also only happen once per socket

4 always remember to set the "error" handler to prevent uncaught exceptions

Fire up the server and connect to it with `netcat/telnet` and play around with it a bit. You should see the `console.log()` calls for the events being printed to the server's stdout as you mash around on the keyboard in the client app.

Now that you can build low-level TCP servers in Node, you're probably wondering how to write a client program in Node to interact with these servers. Let's do that now.

12.2.3 Creating a TCP client

Node isn't only about server software; creating client networking programs is equally as useful, and just as easy in Node. The heart of creating raw connections to TCP servers is the `net.connect()` function. This function accepts an options argument with `host` and `port` values and returns a `Socket` instance. The `socket` returned from `net.connect()` starts off disconnected from the server. So usually you wait for the "connect" event before doing any work with the socket:

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

var socket = net.connect({ port: 1337, host: 'localhost' });
socket.on('connect', function () {
  // begin writing your "request"
  socket.write('HELO local.domain.name\r\n');
  ...
});
```

Once the `socket` instance is connected to the server then it behaves identically to the `socket` instances that you get inside of a `net.Server` callback function.

Let's demonstrate by writing a basic replica of the `netcat(1)` command, implemented in listing 12.6. Basically the program connects to the specified remote server and pipes `stdin` from the program to the socket, and pipes the socket's response to the program's `stdout`.

Listing 12.6 netcat.js: A basic replica of the `netcat(1)` command implemented using Node.js

```
var net = require('net');
var host = process.argv[2];
var port = Number(process.argv[3]); ①

var socket = net.connect(port, host); ②

socket.on('connect', function () { ③
  process.stdin.pipe(socket);
  socket.pipe(process.stdout); ④
  process.stdin.resume(); ⑤
}); ⑥

socket.on('end', function () { ⑦
```

```
    process.stdin.pause();
});
```

- ➊ parse the "host" and "port" from the command-line arguments
- ➋ create the "socket" instance, begin connecting to the server
- ➌ "connect" is emitted once the connection to the server is established
- ➍ pipe the process' stdin to the socket
- ➎ pipe the socket's data to the process' stdout
- ➏ we must call "resume()" on stdin to begin reading data
- ➐ when the "end" event happens, then pause stdin

You will be able to connect to the TCP server examples you wrote before, or for you Star Wars fans, try invoking our netcat replica script with the following arguments for a special easter egg:

```
$ node netcat.js towel.blinkenlights.nl 23
```

Sit back and enjoy the special easter egg show (figure 12.3). You deserve a break:



Figure 12.3 Connecting to the ASCII Star Wars server with the `netcat.js` script

So really that's all it takes to write low-level TCP servers and clients using Node.js. The `net` module provides a simple, yet comprehensive API, and the `Socket` class follows both the readable and writable Stream interfaces as you would expect. Essentially, the `net` module is a showcase of the core fundamentals of Node.

Let's switch gears once again, and look at Node's core APIs that allow you to interact with the process' environment and query information about the runtime and operating system.

12.3 Tools for interacting with the operating system environment

Often times you will find yourself in situations where you want to interact with the environment that Node is running in. Some examples of this could be checking environment variables to enable debug-mode logging, implementing a Linux joystick driver using the low-level `fs` functions to interact with `/dev/js0` (the device file for a game joystick), or launching an external child process like `php` to compile a legacy PHP script to serve to your HTTP clients.

Doing any of these kinds of actions requires using some of the Node core APIs, and we will cover some of these modules throughout this section:

- The global `process` object, which contains information about the current process like the arguments given to it and the environment variables that are currently set.
- The `fs` module, which contains the high level `ReadStream` and `WriteStream` classes which you are familiar with by now, but also houses the low-level functions, which is what we will go over.
- The `child_process` module, which contains both a low-level and high-level interface for spawning child processes, and well as a special way to spawn `node` instances with a 2-way message-passing channel.

The `process` object is one of those APIs that a large majority of programs will interact with, so let's start with that.

12.3.1 The Process global singleton

Every Node process has a single global `process` object that every module shares access to. Various useful information about the process and the context it's running in can be found on this object. For example, the arguments that were invoked with Node to run the current script can be accessed with `process.argv`, or the environment variables can be get or set using the `process.env` object. But the most interesting feature of the `process` object is that it's an `EventEmitter` instance, which has very special events emitted on it like `"exit"` and `"uncaughtException"`.

The `process` object has a lot of bells and whistles under the hood, and we will cover some of the APIs not mentioned in this section later in the chapter. Here though, you will learn about:

- Using `process.env` to get and set environment variables.
- Listening for special events emitted by `process`, namely `"exit"` and `"uncaughtException"`.
- Listening for signal events emitted by `process`, like `"SIGUSR2"` and `"SIGKILL"`.

USING `PROCESS.ENV` TO GET AND SET ENVIRONMENT VARIABLES

Environment variables are a great way to alter the way your program or module will work. For example, you can use these variables as configuration for your server, specifying which port to listen on. Another example is that the operating system can set the TMPDIR variable to specify where your programs should output temporary files that may be cleaned up later.

NOTE

Environment Variables?

In case you're not already familiar with what exactly "environment variables"¹⁴ are, they are simply a set of key-value pairs that any given process can use to affect the way it will behave. For example, all operating systems use the PATH environment variable as a list of file paths to search when looking up a program's location by name (i.e. ls being resolved to /bin/ls).

Footnote 14 http://wikipedia.org/wiki/Environment_variable

Suppose you wanted to enable special debug-mode logging while developing or debugging your module, but not during regular usage, since that would be annoying for consumers of your module. A great way to do this is with environment variables. You could check what the DEBUG variable is set to by checking process.env.DEBUG and seeing if it's set to a truthy value, like shown in listing 12.7.

Listing 12.7 debug-mode.js: Conditionally defining a "debug" function depending on what the DEBUG environment variable is set to

```
var debug;
if (process.env.DEBUG) {    ①
  debug = function (data) {
    console.error(data);    ②
  };
} else {
  debug = function () {};  ③
}

debug('this is a debug call'); ④
console.log('Hello World!');
```

```
debug('this another debug call'); ⑤
```

- ① The first step is to set the "debug" function based on process.env.DEBUG
- ② When it is set, then the "debug" function will log the given argument to stderr
- ③ When it's not set, then the "debug" function is empty and does nothing
- ④ Your script calls the "debug" function in various places throughout your code
- ⑤ This is a nice way to get diagnostic reporting to stderr when debugging a problem in your code

Now if you try running this script regularly (without the process.env.DEBUG environment variable set) then you will see that calls to "debug" do nothing, since the empty function is being called.

```
$ node debug-mode.js
Hello World!
```

To test out "debug mode" you need to set the process.env.DEBUG environment variable. The simplest way to do this when launching a Node instance is to prepend the command with DEBUG=1. When in debug mode, then calls to the "debug" function will be printed to the console as well as the regular output.

```
$ DEBUG=1 node debug-mode.js
this is a debug call
Hello World!
this is another debug call
```

The community module `debug`¹⁵ by TJ Holowaychuk encapsulates precisely this functionality with some additional features. If you like the debugging technique presented here then you should definitely check it out!

Footnote 15 <https://github.com/visionmedia/debug>

SPECIAL EVENTS EMITTED BY `PROCESS`

Normally there are two special events that get emitted by the `process` object:

- "exit" which gets emitted right before the process exits.
- "uncaughtException" which gets emitted any time an unhandled Error is thrown.

The "exit" event is essential for any application that needs to do something right before the program exits, like clean up an object or print a final message to the

console. One important thing to note is that the "exit" event gets fired after the event loop has already stopped, therefore you do not have the opportunity to start any asynchronous tasks during the "exit" event. The exit code is passed as the first argument, which is 0 on a successful exit. Let's write a script that listens on the "exit" event to print an "Exiting..." message:

```
process.on('exit', function (code) {
  console.log('Exiting...');
});
```

The other special event emitted by process is the "uncaughtException" event. In the "perfect program" there will never be any uncaught exceptions. But in the real world we all know that this is impossible, and it's better to be safe than sorry. The only argument given to the "uncaughtException" event is the uncaught Error object.

When there are no listeners for "uncaughtException" then any uncaught Errors will crash the process (this is the default behavior for most applications), but when there's at least one listener then it's up to that listener to decide what to do with the Error and Node will not exit automatically, though it is considered mandatory to do so in your own callback. The Node.js documentation explicitly warns that any usage of this event should contain a `process.exit()` call within the callback, otherwise you leave the application in an undefined state which is bad. Let's listen for "uncaughtException", then throw an uncaught Error to see it in action:

```
process.on('uncaughtException', function (err) {
  console.error('got uncaught exception:', err.message);
  process.exit(1);
});

throw new Error('an uncaught exception');
```

CATCHING SIGNALS SENT TO THE PROCESS

Unix has the concept of "signals", which are a basic form of inter-process communication (IPC¹⁶). These signals are very primitive, allowing for only a fixed set of names to be used, and no arguments to be passed. Node has default behavior for a few signals, which we will go over now:

Footnote 16 http://wikipedia.org/wiki/Inter-process_communication

- **SIGINT** - Gets sent by your shell when you press Ctrl+C. Node's default behavior is to kill the process, but this can be overridden with a single listener for "SIGINT" on `process`.
- **SIGUSR1** - When this signal is received, Node will enter its built-in debugger.
- **SIGWINCH** - Gets sent by your shell when the terminal is resized. Node resets `process.stdout.rows` and `process.stdout.columns` and emits a "resize" event when this is received.

So those are the three signals that Node handles for you by default, but you can also listen for any of these events and invoke a callback function by listening for the event name of the `process` object.

Say you have your server that you are writing. However, when you press Ctrl+C to kill the server, it is an unclean shutdown, and any pending connections will be dropped. The solution to this would be to catch the "SIGINT" signal and stop the server from accepting connections, letting any existing connections complete before the process completes. This is done by simply listening for `process.on('SIGINT', ...)`. The name of the event emitted is the same as the signal name.

```
process.on('SIGINT', function () {
  console.log('Got Ctrl+C!');
  server.close();
});
```

Now, when you press Ctrl+C on your keyboard then the "SIGINT" signal will be sent to the Node process from your shell, which will invoke the registered callback instead of killing the process. Since the default behavior of most applications is to exit the process, it is usually a good idea to do the same in your own "SIGINT" handler, after any necessary "shut down" actions happen. In this case, stopping a server from accepting connections will do the trick. This also works on Windows, despite its lack of proper signals, due to libuv handling the equivalent Windows actions and simulating artificial signals in node.

You can apply this same technique to catch any of the Unix signals¹⁷ that get sent to your Node process. Unfortunately signals in general don't work on Windows, except for the few simulated supported signals like "SIGINT" mentioned before.

Footnote 17 http://wikipedia.org/wiki/Unix_signal#POSIX_signals

12.3.2 Using the Filesystem module

The "fs" module provides all the different functions for interacting with the filesystem of the computer that node is running on. Most of the functions are 1-1 mappings to their C function counterparts, but there are also higher level abstractions like `fs.readFile()` and `fs.writeFile()`, and the `fs.ReadStream` and `fs.WriteStream` classes, which build on top of `open()`, `read()/write()`, and `close()`.

Nearly all of the low-level functions are identical in function to their C versions of the functions. In fact, most of the Node documentation simply says to refer to the equivalent man page explaining the matching C function. You can easily identify these low-level functions because they will always have a synchronous counterpart. For example, `fs.stat()` and `fs.statSync()` are the low-level bindings to the `stat(2)` C function.

NOTE

Synchronous functions in Node.js

As you already know, Node.js is all about asynchronous functions and never blocking the event loop. So why bother including synchronous versions of these filesystem functions? The answer to that is because of Node's own module system, where the `require()` function is synchronous. Since that is implemented using the `fs` module functions, then synchronous counterparts were necessary. However, the golden rule is that they should *only* be used during startup, or when your module is initially loaded, and *never* after that.

Let's take a look at some examples of interacting with the filesystem.

MOVING A FILE

A seemingly simple, yet very common task, when interacting with the filesystem is the act of moving a file from one directory to another. On Unix platforms you use the `mv` command for this, on Windows it's the `move` command. So doing the same thing in Node should be similarly simple, you are probably thinking.

Well if you browse through the `fs` module in the REPL or in the documentation¹⁸, you'll notice that there's no `fs.move()` function. However there is an `fs.rename()` function, which is the same thing if you think about it. Perfect!

Footnote 18 <http://nodejs.org/api/fs.html>

Not so fast there cowboy. `fs.rename()` maps directly to the `rename(2)` C function. One gotcha with this function is that it doesn't work across physical devices (like two hard drives). So this would not work properly, and would throw an EXDEV error:

```
fs.rename('C:\\hello.txt', 'D:\\hello.txt', function (err) {
  // err.code === 'EXDEV'
});
```

So what do you do now? Well you can still create new files on D:\\, and read files from C:\\, so copying the file over will work. With this knowledge, you can create an optimized `move()` function. By optimized we mean that it 1) calls the very fast `fs.rename()` when possible and 2) works across devices by copying the file over in that case using `fs.ReadStream` and `fs.WriteStream`. One such implementation is shown in listing 12.8.

Listing 12.8 copy.js: An optimized "move()" function that renames when possible (fast), and falls back to copying (slow)

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

module.exports = function move (oldPath, newPath, callback) {
  fs.rename(oldPath, newPath, function (err) { ①
    if (err) {
      if (err.code === 'EXDEV') { ②
        copy();
      } else {
        callback(err); ③
      }
      return;
    }
    callback(); ④
  });
}

function copy () {
  var readStream = fs.createReadStream(oldPath); ⑤
  var writeStream = fs.createWriteStream(newPath);
  readStream.on('error', callback);
  writeStream.on('error', callback);
  readStream.on('close', function () {
    fs.unlink(oldPath, callback); ⑥
  });
}
```

```

        readStream.pipe(writeStream);
    }
}

```

- 1 First step is to try calling `fs.rename()` and hope it works
- 2 If we got an EXDEV error, then fall back to the "copy" technique
- 3 Any other kind of error we fail and report back to the caller
- 4 If `fs.rename()` worked successfully, then we are done. This is the optimal case.
- 5 The "copy" technique reads the original file and pipes it out to the destination path
- 6 Once the copy is done, then "unlink" (delete) the original file

You can test out this module directly in the node REPL if you like:

```

$ node
> var move = require('./copy')
> move('copy.js', 'copy.js.bak', function (err) { if (err) throw err })

```

Note that this only works with files, and not directories. To make directories work you would have to first check if the given path is a directory, and if it is then call `fs.readdir()` and `fs.mkdir()` as necessary, but you can implement that at some point in the future on your own.

NOTE

`fs` module error codes

The `fs` module returns standard Unix names¹⁹ for the file system error codes, thus some familiarity with those names is required. These names get normalized, even on Windows, by libuv so that your application only needs to check for one error code at a time. According to the gnu docs page, an EXDEV error happens when "an attempt to make an improper link across file systems was detected."

Footnote 19

http://www.gnu.org/software/libc/manual/html_node/Error-Codes.html

WATCHING A DIRECTORY OR FILE FOR CHANGES

`fs.watchFile()` has been around since the early days. It is expensive on some platforms because it uses polling to see if the file has changed. That is, it `stat()`s the file, waits a short period of time, then `stat()`s again in a continuous loop, invoking the watcher function any time the file has changed.

So say you are writing a module that logs changes from the `systemd` log file. To do this you would want to have a callback function be invoked any time the global "system.log" file was modified.

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

fs.watchFile('/var/log/system.log', function (curr, prev) {
  if (curr.mtime.getTime() !== prev.mtime.getTime()) {
    console.log('"system.log" has been modified');
  }
});
```

The `curr` and `prev` variables are the previous and current `fs.Stat` objects, which should have a different timestamp for one of the file times attached. In this example, the "mtime" values are being compared, as you only want to be notified when the file is modified, and not when it is simply accessed.

`fs.watch()` was introduced in the Node v0.6 release. As we mentioned earlier, it is actually more optimized because, underneath the hood, it uses the platform's native file change notification API for watching files. Because of this, the function is also capable of watching directories, and have its callback function be invoked for a change to any file in the directory. In practice `fs.watch()` is less reliable though, because of various differences between the platforms' underlying file watching mechanisms. For example, the filename parameter does not get reported on OS X when watching a directory, and it is up to Apple to change that in a future release of OS X.

USING COMMUNITY MODULES: "FSTREAM" AND "FILED"

So the `fs` module, like all of Node's core API as you have learned, sticks to being low-level. This means that there is plenty of room to innovate and create awesome abstractions on top of it. Node's active collection of modules is growing on npm every day, and as you could guess there are some quality ones out there that extend the `fs` module to go even further.

"`fstream`"²⁰ by Isaac Schlueter, is one of the core pieces of npm itself. This

module is interesting because it began life as a part of npm, and then got extracted, because its general-purpose functionality is useful to many kinds of command-line applications and sysadmin scripts. One of the awesome features that sets `fstream` apart is its seamless handling of permissions and symbolic links when dealing with directories, which are all automatically transferred over by default.

Footnote 20 <https://github.com/isaacs/fstream>

Using `fstream`, the equivalent of `cp -rp sourceDir destDir` (copying a directory and its contents recursively, and transferring over ownership and permissions) involves simply piping a Reader instance to a Writer instance. In the example here, we're also utilizing the "filter" feature of `fstream` to conditionally exclude files based on a callback function.

```
fstream
  .Reader("path/to/dir")
  .pipe(fstream.Writer({ path: "path/to/other/dir", filter: isValid })

// checks the file that is about to be written and
// returns whether or not it should be copied over
function isValid () {
  // ignore temp files from text editors like TextMate
  return this.path[this.path.length - 1] !== '~';
}
```

"`filed`"²¹ by Mikeal Rogers, is another influential module, mostly because it is written by the same author as the highly popular module, `request`. These modules made popular a new kind of flow control over Stream instances: listening for the "pipe" event, and acting differently based on what is being piped to it (or what it is being piped to).

Footnote 21 <https://github.com/mikeal/filed>

To demonstrate the power of this, take a look at how `filed` turns a regular HTTP server into a full-featured static file server with just one line of code!

```
http.createServer(function (req, res) {
  req.pipe(filed('path/to/static/files')).pipe(res);
});
```

This actually takes care of sending "Content-Length", sending the proper caching headers. In the case where the browser already has the file cached, then

`filed` will actually respond to the HTTP request with a 304 response, skipping the whole opening and reading the file from the disk process. These are the kinds of optimizations that acting on the "pipe" event make possible, because the `filed` instance has access to both the `req` and `res` objects of the HTTP request.

We have only demonstrated two examples of good community modules that extend the base `fs` module to do awesome things or expose beautiful APIs. The `npm search` command is a good way to find published modules for a given task. Say you wanted to find another module that simplifies copying files from one destination to another. Executing `npm search copy` could bring up some useful results in this case. When you find a published module that looks interesting, then you can execute `npm info module-name` to get information about the module like its description, homepage and published versions. Just remember that for any given task, it's very likely that someone has attempted to solve the problem in the form of an `npm` module, so always check there before writing something from scratch.

12.3.3 Spawning external processes

Node provides the `child_process` module to create child subprocesses from within a Node server or script. There are two APIs for this: a high-level one, called `exec()`, and a low-level one, called `spawn()`. Either one may be appropriate for you to use depending on your needs. There is also a special way to create child processes of Node itself, with a special Inter-Process Communication (IPC) channel built-in called `fork()`. All of these functions are meant for different use-cases which we will go over now.

- `cp.exec()` - the high level API for spawning commands and buffering the result into a callback
- `cp.spawn()` - the low level API for spawning single commands into a `ChildProcess` object
- `cp.fork()` - the special way to spawn additional Node processes with a built-in IPC channel

NOTE**Pros and cons to child processes**

There are great benefits to using child processes but there are downsides as well. The obvious one is that the program being executed needs to actually be installed on the user's machine, making it a dependency of your application. The alternative would be to do whatever the child process does in JavaScript itself. A good example of this is `npm`, which once upon a time used the system `tar` command when extracting node packages. But even though `tar` is available on every Unix computer out there, there were many conflicts relating to incompatible versions of tar itself, and on top of that it's very rare for a Windows computer to have installed. These factors, as well as the importance for `npm` to work everywhere, lead to `node-tar`²² being written entirely in JS, not using any child processes. Conversely, you would never want to re-implement `php` if you were serving PHP scripts, so that's a case where child processes are good.

Footnote 22 <https://github.com/isaacs/node-tar>

BUFFERING COMMAND RESULTS USING CP.EXEC()

The high level API, `cp.exec()`, is useful for when you want to invoke a command, and only care about the final result, and don't care about accessing the data from child's stdio streams as they come. Another benefit is that you can enter full sequences of commands, including multiple processes being piped to one another.

One good use-case for the `exec()` API is when you are accepting user commands to execute. Say you are writing an IRC bot, and would like to execute commands when any user says something beginning with ". ". So if a user typed ".ls" as their IRC message, then the bot would execute `ls` and print the output back to the IRC room. Be sure to set the `timeout` option, so that any neverending processes get automatically killed after a certain period of time, like shown in listing 12.9.

Listing 12.9 room.js: Using cp.exec() to spawn user-given commands from an IRC room bot

```
var cp = require('child_process');
```

```

room.on('message', function (user, message) { ①
  if (message[0] === '.') { ②
    var command = message.substring(1); ③
    cp.exec(command, { timeout: 15000 },
      function (err, stdout, stderr) {
        if (err) { ④
          room.say(
            'Error executing command "' + command + '": ' + err.message
          );
          room.say(stderr);
        } else {
          room.say('Command completed: ' + command);
          room.say(stdout);
        }
      }
    );
  }
});

```

- ① "room" is an object representing a connection to an IRC room (from some theoretical irc module)
- ② The "message" event gets emitted for each IRC message sent to the room
- ③ This script checks the message contents and checks if it begins with "."
- ④ Spawn a child process and have node buffer the result into a callback for us, timing out after 15 seconds

There are some good modules already in the npm registry implementing the IRC protocol, so if you would like to write an IRC bot like in this example for real, then you should definitely use one of the existing modules (both `irc` or `irc-js` in the npm registry are popular).

For times when you need to buffer a command's output, but would like node to automatically escape the arguments for you, there's the `execFile()` function. This function takes four arguments, rather than three, and you pass the executable to run, along with an array of arguments to invoke the executable with. This is useful when you have to incrementally build up the arguments that the child process is going to use:

```

cp.execFile('ls', [ '-l', process.cwd() ],
  function (err, stdout, stderr) {
  if (err) throw err;
  console.error(stdout);
}

```

```
});
```

SPAWNING COMMANDS WITH A STREAM INTERFACE USING CP.SPAWN()

The low level API for spawning child processes in Node is `cp.spawn()`. This function differs from `cp.exec()` because it returns a `ChildProcess` object which you interact with. Rather than giving it a single callback function when the process completes, `cp.spawn()` lets you interact with each stdio stream of the child process as individual `Stream` instances, which include all the usual benefits of Streams in Node.

The most basic use of `cp.spawn()` looks like:

```
var child = cp.spawn('ls', [ '-l' ]);

// stdout is a regular Stream instance, which emits 'data',
// 'end', etc.
child.stdout.pipe(fs.createWriteStream('ls-result.txt'));

child.on('exit', function (code, signal) {
  // emitted when the child process exits
});
```

The first argument is the program to execute. This can be a single program name, which will be looked up in the current PATH, or an absolute path to a program to invoke. The second argument is an Array of string arguments to invoke the process with. In the default case, a `ChildProcess` object contains 3 built-in `Stream` instances which your script is meant to interact with:

- `child.stdin` is the *writable* Stream that represents the child's `stdin`.
- `child.stdout` is the *readable* Stream that represents the child's `stdout`.
- `child.stderr` is the *readable* Stream that represents the child's `stderr`.

You can do whatever you want with these streams, like pipe them to a file or socket or some other kind of writable stream. You can even just completely ignore them if you'd like.

The other interesting event that happens on `ChildProcess` objects is the "exit" event, which is fired when the process has exited and the associated stream objects have all ended.

One good example module that abstracts the use of `cp.spawn()` into helpful

functionality is "node-cgi"²³, which allows you to reuse legacy CGI scripts in your Node HTTP servers. Older style websites were written using the "Common Gateway Interface", which was really just a standard for responding to HTTP requests by invoking arbitrary "CGI scripts" as child processes of an HTTP server with special environment variables describing the request. For example, you can write a CGI script that uses `sh` as the CGI interface:

Footnote 23 <https://github.com/TooTallNate/node-cgi>

```
#!/bin/sh
echo "Status: 200"
echo "Content-Type: text/plain"
echo
echo "Hello $QUERY_STRING"
```

If you were to name that file "hello.cgi" (don't forget to `chmod +x hello.cgi` to make it executable!), then you could easily invoke it as the response logic for HTTP requests in your HTTP server with a single line of code.

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

var server = http.createServer( cgi('hello.cgi') );
server.listen(3000);
```

With this server setup, when an HTTP request hits the server, then `node-cgi` will handle the request by:

- Spawning the "hello.cgi" script as a new child process using `cp.spawn()`.
- Passing the new process contextual information about the current HTTP request using a custom set of environment variables.

The "hello.cgi" script uses one of the CGI-specific environment variables, `QUERY_STRING`, which contains the query-string portion of the request URL, which the script uses in the response which gets written to the script's `stdout`. If you were to fire up this example server and send an HTTP request to it using `curl`, you will see something like:

```
$ curl http://localhost:3000/?nathan
Hello nathan
```

There are a lot of very good use cases for child processes in Node. `node-cgi` is just one example of them, but you will find that while getting your server or application to do what it needs to do, you will inevitably have to utilize them at some point.

DISTRIBUTING THE WORKLOAD USING CP.FORK()

The last API offered by the `child_process` module is a specialized way of spawning additional Node processes, but with a special IPC channel built in. Since you're always spawning Node itself, the first argument passed to `cp.fork()` is a path to a Node.js module to execute.

Like `cp.spawn()`, this function returns a `ChildProcess` object. However the major difference is the API added by the IPC channel. So, the child process now has a `child.send(message)` function, and the script being invoked by `fork()` can listen for `process.on('message')` events.

So say you wanted to write a Node HTTP server that calculated the fibonacci sequence. You may try naively writing the server all in one shot, like shown in listing 12.10:

Listing 12.10 fibonacci-naive.js: A non-optimal implementaion of a fibonacci HTTP server in Node.js

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

function fib (n) { ①
  if (n < 2) {
    return 1;
  } else {
    return fib(n - 2) + fib(n - 1);
  }
}

var server = http.createServer(function (req, res) { ②
  var num = parseInt(req.url.substring(1), 10);
  res.writeHead(200);
  res.end(fib(num) + "\n");
});
server.listen(8000);
```

① Fibonacci number calculation function

② This basic HTTP server simply calculates the requested fibonacci number

If you fire up the server with `node fibonacci-naive.js` and send an HTTP request to `http://localhost:8000` then it will work as expected, but calculating the fibonacci sequence for a given number is an expensive, CPU-bound computation, and while your Node server's single thread is grinding away at calculating the result there, no additional HTTP requests can be served during this time. Additionally, you're only utilizing one CPU core here, where you likely have more than that which are just sitting there doing nothing. This is bad.

`cp.fork()` offers a clean interface for us to do this in Node. The trick is to fork Node processes during each HTTP request and have the child process do the expensive calculation, and report back to the server using fork's IPC channel. Doing this will take 2 files:

- "fibonacci-server.js" will be the server.
- "fibonacci-calc.js" does the calculation.

First the server:

```
var http = require('http');
var cp = require('child_process');

var server = http.createServer(function(req, res) {
  var child = cp.fork(__filename, [ req.url.substring(1) ]);
  child.on('message', function(m) {
    res.end(m.result + '\n');
  });
});
server.listen(8000);
```

So now the server is using `cp.fork()` to have the fibonacci calculation logic be done in a separate Node process, which will report back to the parent process using `process.send()` as shown below in the "fibonacci-calc.js" script:

```
function fib(n) {
  if (n < 2) {
    return 1;
  } else {
    return fib(n - 2) + fib(n - 1);
  }
}
```

```
var input = parseInt(process.argv[2], 10);
process.send({ result: fib(input) });
```

This time you can start the server with `node fibonacci-server.js` and, again send an HTTP request to `http://localhost:8000`. This is just one great example of how dividing up the various components that make your application into multiple processes can be a great benefit to you, and `cp.fork()` provides the perfect communication interface with `child.send()` and `child.on('message')` for the parent, and `process.send()` and `process.on('message')` for the child. Use them!

12.4 Developing command-line tools

Another very common task fulfilled by Node scripts is to build command-line tools. By now you should be familiar with the largest command-line tool written in 100% Node: "node package manager", a.k.a npm. As a package manager, it does a lot of filesystem operations and spawning of child processes, and all of this is done using Node and its asynchronous APIs. npm takes advantage of this by installing packages in parallel, rather than serially, making the overall process faster. So if a command-line tool *that* complicated can be written in Node, then anything can.

Most command-line programs have common process-related needs, like parsing command-line arguments used, and reading from `stdin` and writing to `stdout` and `stderr`. In this section, you will learn about the common requirements for writing a full command-line program, including:

- Parsing the command-line arguments
- Working with the `stdin` and `stdout` Streams
- Adding pretty colors to the output using `ansi.js`

To get started on building awesome command-line programs, you need to be able to read the arguments the user invoked your program with. We will take a look at that first.

12.4.1 Parsing command-line arguments

Parsing arguments is a very easy and straightforward process. Node provides you with the `process.argv` property, which is an Array of Strings which are the arguments that were used when Node was invoked. The first entry of the Array is the "node" executable, while the second entry is the name of your script. Now parsing and acting on these arguments simply requires iterating through the Array entries and inspecting each argument.

To demonstrate, write a quick script called `args.js` that simply prints out the result of `process.argv`. Since 90% of the time you don't care about the first 2 entries, you can `slice()` them off before processing.

```
var args = process.argv.slice(2);
console.log(args);
```

Now when you invoke this script standalone, you will get an empty Array, since no additional arguments were passed in:

```
$ node args.js
[]
```

But when you pass along "hello" and "world" as arguments, then the Array contains string values as you would expect:

```
$ node args.js hello world
[ 'hello', 'world' ]
```

And as with any terminal application, you can use quotes for arguments that have spaces in them to combine them into a single argument. This is not a feature of Node, but rather the shell that your are using (likely bash on a Unix platform or cmd.exe on Windows).

```
$ node args.js "tobi is a ferret"
[ 'tobi is a ferret' ]
```

By Unix convention, every command-line program should respond to the `-h` and `--help` flags by printing out usage instructions and then exiting. Listing

12.11 shows an example of using `Array#forEach()` to iterate on the arguments and parse them in the callback, printing out the usage instructions when the expecting switch is encountered.

Listing 12.11 args-parse.js: Parsing `process.argv` using `Array#forEach()` and a switch block.

```
var args = process.argv.slice(2);      ①

args.forEach(function (arg) {          ②
  switch (arg) {
    case '-h':
    case '--help':
      printHelp();
      break;                         ③
  }
});                                ④

function printHelp () {
  console.log('  usage:');
  console.log(' $ AwesomeProgram <options> <file-to-awesomeify>');
  console.log('  example:');
  console.log(' $ AwesomeProgram --make-awesome not-yet.awesome');
  process.exit(0);
}
```

- ① Start off by slicing off the first two entries, which you're not interested in
- ② Iterate through the arguments, looking for -h or --help
- ③ You can add additional flags/switches here that your application responds to
- ④ When "help" is requested, just print out a helpful message then quit

You can easily extend that `switch` block to parse additional switches. Community modules like `commander.js`, `nopt`, `optimist`, and `nomnom` (just to name a few) all attempt to solve this problem in their own way, so don't feel like using a `switch` block is the only way to parse the arguments. In reality, like so many things in programming, there is no single correct way to do it.

Another task that every command line program will need to deal with is reading input from `stdin`, and writing structured data to `stdout`. Let's take a look at how this is done in Node next.

12.4.2 Working with `stdin` and `stdout`

A common Unix idiom is to have programs be small, self-contained, and focused on a single task that they are really good at. This is usually facilitated by using pipes, feeding the results of one process to the next, repeated until the end of the command chain. For example, using standard Unix commands to, say, retrieve the list of unique authors from any given git repository, you could combine the `git log`, `sort` and `uniq` commands, like this:

```
$ git log --format='%an' | sort | uniq
Mike Cantelon
Nathan Rajlich
TJ Holowaychuk
```

These commands run in parallel, feeding the output of the first process to the next, continuing on until the end. To adhere to this "piping" idiom, Node provides two `Stream` objects for your command-line program to work with:

- `process.stdin` - a `ReadStream` that is used to read input data
- `process.stdout` - a `WriteStream` to write output data to.

These objects act like the familiar Stream interfaces that you've already learned about.

WRITING OUTPUT DATA WITH `PROCESS.STDOUT`

You've already been using the `process.stdout` writable stream implicitly every time you have called `console.log()`. Internally, the `console.log()` function calls `process.stdout.write()` after formatting the input arguments. But the `console` functions are more for debugging and inspecting objects, for the times when you need to write structured data to `stdout`, you can call `process.stdout.write()` directly. Say your program connects to an HTTP url and writes the response to `stdout`. Utilizing `Stream#pipe()` works well, as shown here:

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

var target = url.parse(process.argv[2]);
var req = http.get(target, function (res) {
  res.pipe(process.stdout);
});
```

And voilà! An absolutely minimal `curl` replica in only 7 lines of code. Not too bad, huh? Next up let's cover `process.stdin`.

READING INPUT DATA WITH PROCESS.STDIN

You must initially call `process.stdin.resume()` to signify that your script is interested in data from `stdin`. After that it acts like any other readable stream, by emitting "data" events as data is received from the output of another process, or as the user enters keystrokes into the terminal window. Listing 12.12 shows a quick command-line program that prompts the user for their age before deciding to continue executing.

Listing 12.12 stdin.js: An age-restricted program that prompts the user for their age, reading from `stdin`

```

var requiredAge = 18;          ①

process.stdout.write('Please enter your age: ');      ②

process.stdin.setEncoding('utf8');                      ③

process.stdin.on('data', function (data) {
    var age = parseInt(data, 10);                     ④
    if (isNaN(age)) {                                ⑤
        console.log('%s is not a valid number!', data);
    } else if (age < requiredAge) {                   ⑥
        console.log('You must be at least %d to enter, ' +
            'come back in %d years',
            requiredAge, requiredAge - age);
    } else {
        enterTheSecretDungeon();                      ⑦
    }
    process.stdin.pause();                           ⑧
});

process.stdin.resume();                            ⑨

function enterTheSecretDungeon () {
    console.log('Welcome to The Program :)');
}

```

① This example program is age-restricted

- 2 First write a basic question for the user to answer
- 3 Setup stdin to emit utf8 strings instead of Buffers
- 4 Parse the data into a Number
- 5 If the user didn't give a valid number, then print a message saying so
- 6 If the user's given age is less than 18, then print a message saying to come back in a few years
- 7 If those past 2 conditions are ok, then the user is old enough and you can continue executing your program
- 8 This program only waits for a single 'data' event before closing stdin
- 9 process.stdin begins in a paused state, so call resume() to start reading

DIAGNOSTIC LOGGING WITH PROCESS.STDERR

There is also a `process.stderr` writable stream in every Node process, which acts identical to the `process.stdout` stream, except it writes to `stderr` instead. Since `stderr` is usually reserved for debugging, and not so much for sending structured data and piping, you usually end up simply using `console.error()` instead of accessing `process.stderr` directly.

So now that you are familiar with the built-in stdio Streams in Node, which is crucial knowledge for any command-line program, let's move on to something a bit more colorful (pun intended).

12.4.3 Adding colored output

Lots of command-line tools use colored text to make things easier to distinguish on the screen. Node itself does this in its REPL, as does npm, for its various logging levels. It's a nice bonus feature that any command-line program can easily benefit from, and luckily adding colored output to your programs is rather easy, especially with the support of community modules.

CREATING AND WRITING ANSI ESCAPE CODES

Colors on the terminal happen through something called ANSI escape codes (the name comes from the American National Standards Institute). These "escape codes" are simple text sequences written to the `stdout` that have special meaning to the terminal, signifying to change the text color, change the position of the cursor, make a beep sound, and more interactions. Let's start simple. To print the word "hello" in the color green in your script, a single `console.log()` call is all it takes:

```
console.log('\033[32mhello\033[39m');
```

If you look closely, you can see the word "hello" in the middle of the string

with some weird looking characters on either side. This may look confusing at first but it's rather simple really. Figure 12.4 breaks up the green "hello" string into its three distinct pieces:

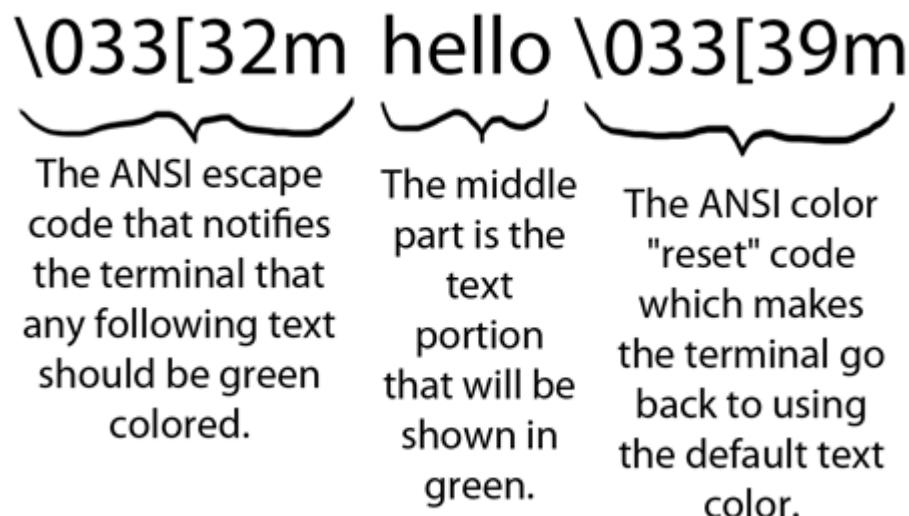


Figure 12.4 "hello" in green text using ANSI escape codes

Now, there are a lot of escape codes that terminals recognize and most developers have better things to do with their time than memorize them all. Thankfully, the Node community comes in to the rescue once again with multiple modules to pick from, like `colors.js`, `cli-color`, and `ansi.js` to name a few, which all try to make using colors in your programs easy and fun.

NOTE

ANSI escape codes on Windows

Technically, Windows and its command prompt (`cmd.exe`) don't support ANSI escape codes. Fortunately for us, Node interprets the escape codes on Windows when your scripts write them to `stdout` and then calls the appropriate Windows functions that would do the equivalent action or color. This is just an interesting tidbit, and not something you'll have to think about while writing your Node application.

FORMATTING FOREGROUND COLORS USING ANSI.JS

Let's take a look at `ansi.js`²⁴ (`npm install ansi`). This module is nice because it's a very thin layer on top of the raw ANSI codes, which gives you greater flexibility compared to the other color modules, which only work with a single string at a time. In `ansi.js`, you set the "modes" (like "bold") of the Stream, which are persistent until cleared by one of the `reset()` calls. And as an added bonus, `ansi.js` is the first module to support 256 color terminals, and has capabilities to convert CSS color codes (such as `#FF0000`) into ANSI color codes.

Footnote 24 <https://github.com/TooTallNate/ansi.js>

`ansi.js` works with the concept of a "cursor", which is really just a wrapper around a writable Stream instance with lots of convenience functions to write ANSI codes to the Stream, all of which support chaining. To print the word "hello" in green text again using `ansi.js` syntax, you would write:

```
var ansi = require('ansi');
var cursor = ansi(process.stdout);

cursor
  .fg.green()
  .write('Hello')
  .fg.reset()
  .write('\n');
```

You can see here that to use `ansi.js` you first have to create a `cursor` instance from a writable Stream. Since you're interested in coloring your program's output, you pass `process.stdout` as the writable stream that the `cursor` will use. Once you have the `cursor` you can invoke any of the methods it provides to alter the way that the text output will be rendered to the terminal. In this case the result is equivalent to the `console.log()` call from before, we're calling:

- `cursor.fg.green()` to set the foreground color to green
- `cursor.write('Hello')` writes the text "Hello" to the terminal in green
- `cursor.fg.reset()` resets the foreground color back to the default
- `cursor.write('\n')` to finish up with a newline

FORMATTING BACKGROUND COLORS USING ANSI.JS

Background colors are also supported. To set the background color instead of the foreground color with `ansi.js` you simply replace the `fg` portion of the call with `bg`. So for example, to set a red background color you would call `cursor.bg.red()`. Let's wrap up with a quick program that prints this book's title all colorful to the terminal like shown in figure 12.5.

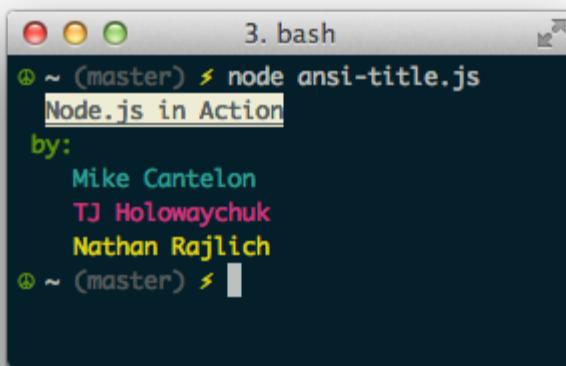


Figure 12.5 The result of the "ansi-title.js" script printing out the name of this book and the authors in different colors

The code to output fancy colors like this is verbose, but very straightforward, since each function call maps 1 to 1 to the corresponding escape code being written to the stream. The example program, shown in listing 12.13, is really just the 2 lines of initialization at the top followed by one really long chain of function calls that end up writing color codes and strings to `process.stdout`.

Listing 12.13 `ansi-title.js`: A simple program that prints this book's title and authors in pretty colors

```
var ansi = require('ansi');
var cursor = ansi(process.stdout);

cursor
  .reset()
  .write(' ')
  .bold()
```

```

.underline()
.bg.white()
.fg.black()
.write('Node.js in Action')
.fg.reset()
.bg.reset()
.resetUnderline()
.resetBold()
.write(' \n')
.fg.green()
.write(' by:\n')
.fg.cyan()
.write('      Mike Cantelon\n')
.fg.magenta()
.write('      TJ Holowaychuk\n')
.fg.yellow()
.write('      Nathan Rajlich\n')
.reset()

```

Color codes are only one of the key features of `ansi.js`. We haven't even touched on any of the cursor positioning codes, or how to make a beep sound, or hiding and showing the cursor, but you can consult the `ansi.js` documentation and examples to see how that works.

12.5 Summary

Node is primarily designed for general I/O related tasks, and HTTP server are only the most common type of task that Node is used to fulfill. However, as you've learned throughout this chapter, Node is well suited for a large variety of different tasks, such as a command-line interface to your application server, a client program that connects to the ASCII Star Wars server, a program that fetches and displays statistics from the stock market servers... The possibilities are really endless, and only limited by your imagination. Take a look at `npm` or `node-gyp` for a couple complicated examples of command line programs being written using Node. They're a couple great examples to learn from.

In this past chapter we talked about a couple community modules that could aid in development of your next application. In this next chapter we're gonna focus on how you can find these awesome modules throughout the Node community, and how you can contribute the modules you've developed back to the community for feedback and improvements. The social interaction is the exciting stuff!

13

The Node ecosystem

In this chapter

- Finding online help with Node
- Collaborating on Node development using GitHub
- Publishing your work using the Node Package Manager

To get the most out of Node development, it pays to know where to go for help and how to share your contributions with the rest of the community. If you’re already familiar with how to work in Node, you may want to skim this chapter; otherwise, read on.

As with most open source communities, the development of Node and related projects happens via online collaboration. Many developers work together to submit and review code, document projects, and report bugs. When developers are ready to release a new version of Node it’s published on the official Node website. When a release-worthy third-party module has been created, it can be published to the npm repository to make it easy for others to install. Online resources provide the support you need to work with Node and related projects. Figure 13.1 shows how you can use online resources for Node-related development, distribution, and support:

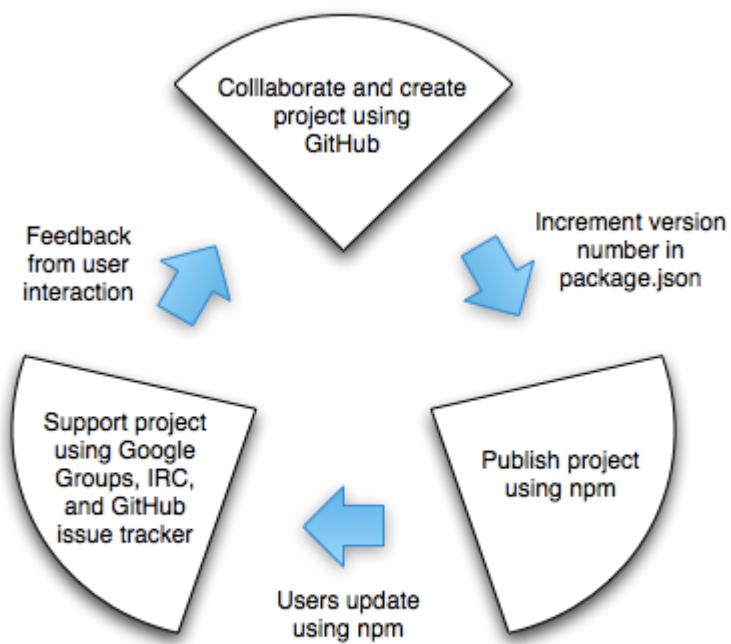


Figure 13.1 Node-related projects are created online collaboratively, often via the GitHub website. They’re then published to npm with documentation and support provided via online resources.

As you’ll likely need support before you need to collaborate, let’s first look at where to go online to get help when you need it.

13.1 *Online resources for Node developers*

As the Node world is an ever-changing one, you’ll find the most up-to-date references online. At your disposal are numerous websites, online discussion groups, and chat rooms where you can find the information you need.

13.1.1 *Node and module references*

Table 13.1 lists a number of Node-related online references and resources. The most useful websites for referencing the Node APIs and learning about available third-party modules are <http://nodejs.org> and <http://npmjs.org>, respectively.

Table 13.1 Useful node.js references.

Resource	URL
Node.js homepage	http://nodejs.org/
Node.js up-to-date core documentation	http://nodejs.org/api/
Node.js blog	http://blog.nodejs.org/
Node.js job board	http://jobs.nodejs.org/
Node Package Manager homepage	http://npmjs.org/

When you attempt to implement something using Node, or any of its built-in modules, the Node homepage is an invaluable resource. The site (shown in figure 13.2) documents the entirety of the Node framework, including each of its APIs. You'll always find documentation for the most recent version of Node on the site. The official blog also documents the latest Node advances and shares important community news. You'll even find a job board.

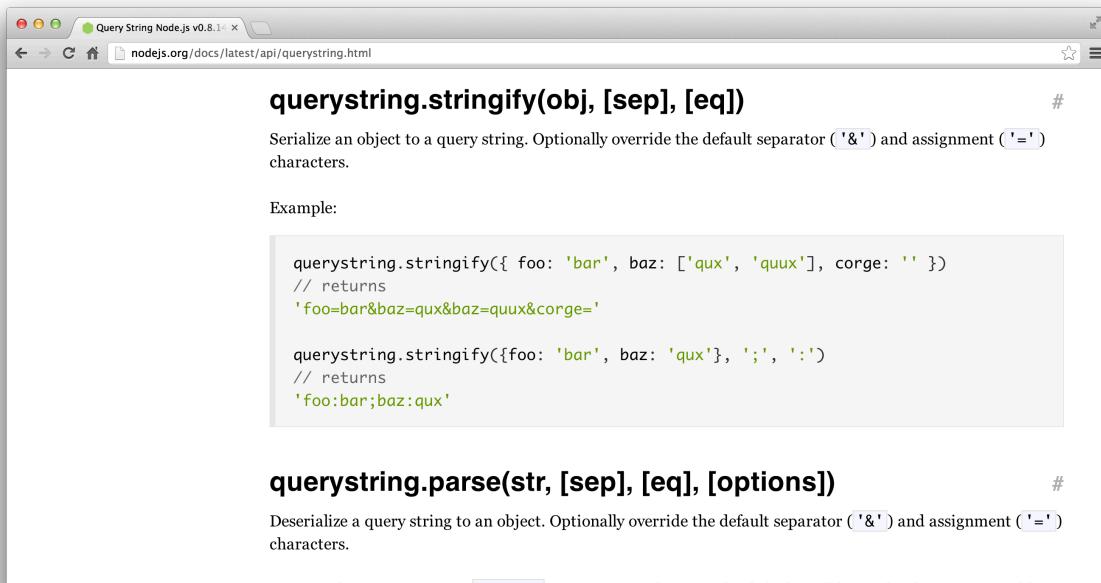


Figure 13.2 In addition to providing links to useful Node-related resources, nodejs.org offers authoritative API documentation for every released version of Node.

When you're shopping for third-party functionality, the npm repository search page is the place to go. It lets you use keywords to search through the thousands of modules available in npm. If you find a module that you'd like to check out, click on the module's name to bring up its detail page, where you'll find links to the module's project homepage, if any, and useful information such as what other npm packages depend on the module, the module's dependencies, which versions of Node the module is compatible with, and its license information.

If you have a question about how to use Node or other third-party modules, these websites may not always help. Luckily, we can point you to some great places to ask for help online.

13.1.2 Google Groups

Google Groups have been set up for Node and some other popular third-party modules, including npm, Express, node-mongodb-native, and Mongoose. Google Groups are useful for tough or in-depth questions. For example, if you were having trouble figuring out how to delete MongoDB documents using the node-mongodb-native module you could go to the node-mongodb-native Google Group¹ and search it to see if anyone else had the same problem. If no one has dealt with your problem, the next step would be to join the Google Group and post your question. You can also write lengthy Google Groups posts, helpful for complicated questions as you have the opportunity to explain your issue thoroughly.

Footnote 1 <https://groups.google.com/forum/?fromgroups#!forum/node-mongodb-native>

But no central list exists that includes all Node-related Google Groups. In order to find them they'll either be mentioned in project documentation or you'll have to search the web. You could, for example, search “nameofsomemodule node.js google group” on Google to check if a Google Group existed for a particular third-party module.

The drawback to using Google Groups is that often you have to wait anywhere from hours to days to get a response, depending on the parameters of the Google Group. For simple questions where you need a quick reply, you should consider entering an internet chat room, where you can often get a quick answer.

13.1.3 IRC

Internet Relay Chat (IRC) was created way back in 1988, and while some think it archaic, it's still alive and active—and is the best online means, in fact, to get answers to quick questions about open source software. IRC rooms are called “channels” and exist for Node and various third-party modules. You won't find a list of Node-related IRC channels anywhere but third-party modules that have a corresponding IRC will sometimes mention them in their documentation.

To get your question answered on IRC, connect to an IRC network², change to the appropriate channel, and send your question to the channel. Out of respect to the folks in the channel, it's good to do some research beforehand to make sure your question can't be solved with a quick web search.

Footnote 2 <http://chatzilla.hacksrus.com/faq/#connect>

If you're new to IRC, the easiest way to get connected is using a web-based

client. Freenode, the IRC network on which most Node-related IRC channel exists, has a web client available at <http://webchat.freenode.net/>. To join a chatroom, enter the appropriate channel into the connection form. You don't need to register and you can enter any nickname (if someone already uses the one you choose, the character ‘_’ will be appended to the end of your nickname to differentiate you).

Once you click “Connect” you'll end up in a chat room with a list in the right sidebar of any other users in the room.

13.1.4 GitHub issues

If a project's development occurs on GitHub, another place to look for solutions to problems with Node modules is the project's GitHub issue queue. To get to the issue queue navigate to the project's main GitHub page and click the “Issues” tab. You can use the search form to look for issues related to your problem. An example issue queue is shown in figure 13.3:

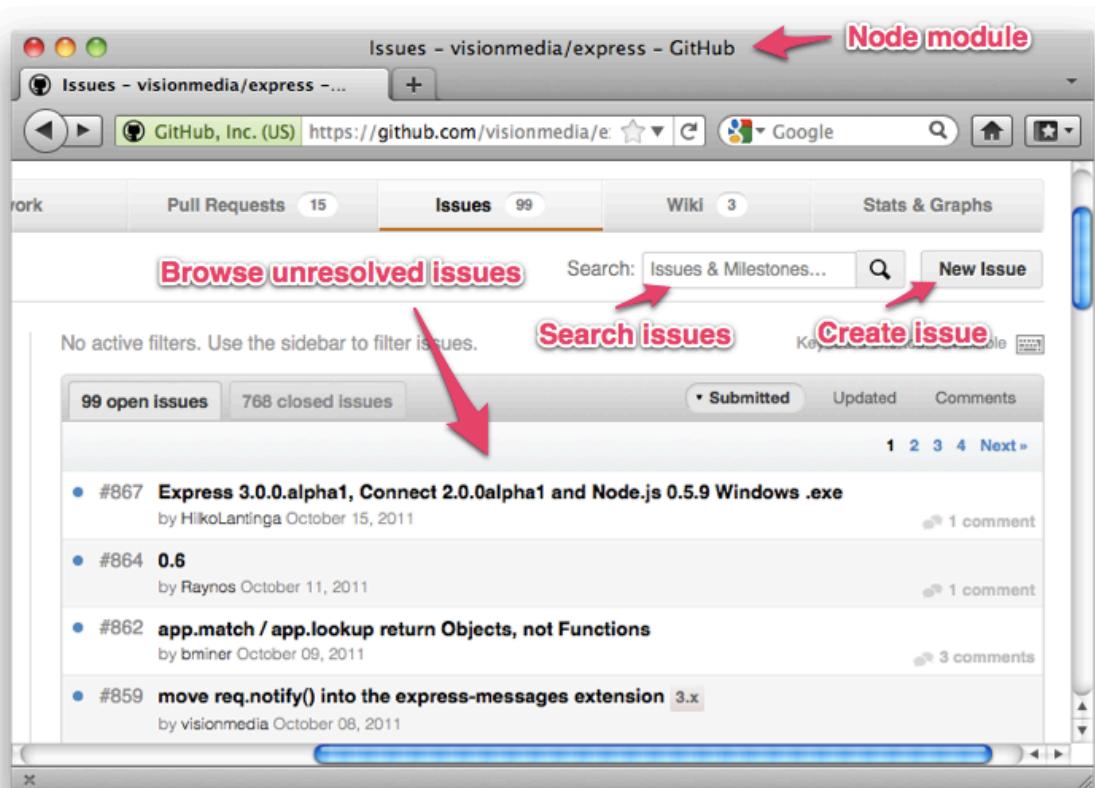


Figure 13.3 For projects hosted on GitHub, the issue queue can be helpful if you believe you've identified a problem in the project's code.

If you're unable to find an issue that addresses your problem, and you think it may be due to a bug in the project's code, then you can click the “New Issue”

button on the issues page to describe the bug. Once you've created an issue the project maintainers will be able to reply on that issue page to address the issue, or possibly ask questions to get a better idea of your problem if they need more information.

NOTE**Issue Tracker is not a Support Forum**

Depending on the project, it may not be considered appropriate for you to open general support questions on the project's GitHub Issue Tracker. This is usually the case if the project has set up another means for users to get general support like a Google Group for example. It's a good idea to read the project's README file to see if it has a preference regarding general support or questions.

Now that you know where to go online to file issues for projects, we're going to talk about GitHub's non-support role as the website through which most Node development collaboration takes place.

13.2 GitHub

GitHub is the center of gravity for much of the open source world and critical to Node developers. The GitHub service provides hosting for Git, a powerful version control system (VCS), and a web interface that allows you to easily browse Git repositories. GitHub is free to use for open source projects.

NOTE**Git**

The Git VCS has become a favorite among open source projects. It's a distributed version control system (DVCS), which, unlike Subversion and many other VCSs, you can use without a network connection to a server. Git was released in 2005, inspired by a another, proprietary, VCS called "BitKeeper." The publisher of BitKeeper had granted the Linux kernel development team free use of the software, but revoked it when suspicion arose that members of the team were attempting to figure out BitKeeper's inner workings. Linus Torvalds, the creator of Linux, decided to create an alternative VCS with similar functionality and, within months, Git was being used by the Linux kernel development team.

In addition to Git hosting, GitHub provides projects with issue tracking, wiki functionality, and web page hosting. As most Node projects in the npm repository are hosted on GitHub, knowing how to use GitHub is helpful for getting the most

out of Node development. GitHub gives you a convenient way to browse code, check for unresolved bugs, and, if need be, contribute fixes and documentation.

Another use of GitHub is to “watch” a project. Watching a project provides you with notification of any changes to the project. The number of people watching a project is often used to gauge a project’s overall popularity.

GitHub may be powerful, but how do you use it? Let’s delve into that next.

13.2.1 Getting started on GitHub

When you’ve come up with an idea for a Node-based project or a third-party module you’ll want to set up an account on GitHub, if you haven’t already, for easy access to Git hosting. After you’re set up you can add your projects, which you’ll learn to do in the next section.

Because GitHub requires use of Git, you’ll want to configure it before continuing to GitHub. Thankfully, GitHub offers help pages for Mac³, Window⁴, and Linux⁵ to help you properly get set up. Once you’ve configured Git, you’ll need to get set up on GitHub by registering on its website and providing a Secure Shell (SSH) public key. You need the SSH key to keep your interactions with GitHub secure. You’ll learn the details of each of these steps in the next section. Note that you only have to do these steps once, not every time you add a project to GitHub.

Footnote 3 <https://help.github.com/articles/set-up-git#platform-mac>

Footnote 4 <https://help.github.com/articles/set-up-git#platform-windows>

Footnote 5 <https://help.github.com/articles/set-up-git#platform-linux>

GIT CONFIGURATION AND GITHUB REGISTRATION

To use GitHub you need to configure your Git tool, letting it know your name and email address. Enter the following two commands:

```
git config --global user.name "Bob Dobbs"
git config --global user.email subgenius@example.com
```

Next, register on the GitHub website. Click to find the sign-up form⁶, fill it in, and click “Create an account.”

Footnote 6 <https://github.com/signup/free>

PROVIDING GITHUB WITH AN SSH PUBLIC KEY

Once you're registered, the next thing you'll need to do is provide GitHub with an SSH public key⁷. You'll use this to authenticate your Git transactions by taking the following steps:

Footnote 7 <http://github.com/guides/providing-your-ssh-key>

1. Click "Account Settings"
2. Click "SSH Keys"
3. Click "Add SSH Key"

At this point what you need to do varies depending on your operating system. GitHub will detect your operating system and show the relevant instructions.

13.2.2 Adding a project to GitHub

Once you're set up on GitHub, you can add a project to your account and begin pushing commits to it. First, you'll create a GitHub repository for your project which we will go over shortly. After that you create a Git repository on your local workstation, which is where you do your work before pushing to the GitHub repository. Figure 13.4 outlines this process, the steps of which we'll cover individually throughout this section. You can also view your project files using GitHub's web interface.

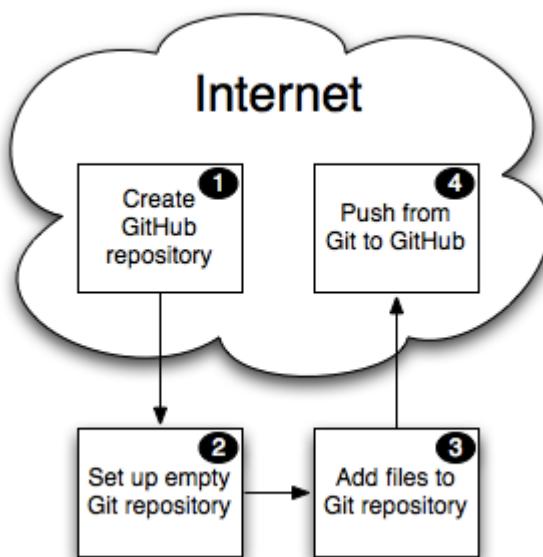


Figure 13.4 The steps to take to add a Node project to GitHub.

CREATING A GITHUB REPOSITORY

Creating a repository on GitHub consists of the following steps:

1. Log in to github.com in your web browser
2. Click on “Dashboard”
3. Click on the “New Repository” icon at the top right
4. Fill out the resulting form describing your repository and click “Create Repository”
5. GitHub then creates an empty Git repository and issue queue for your project
6. You’ll be presented with the steps you need to take to use Git to push your code to GitHub

Because it’s helpful to understand what each of these steps does, in this section we’ll run through an example and you’ll learn the bare essentials of using Git.

SETTING UP AN EMPTY GIT REPOSITORY

To add an example project to GitHub you’re going to make a Node module with URL-shortening logic called “node-elf.” First, create a temporary directory in which to put your project using the following commands:

```
mkdir -p ~/tmp/node-elf
cd ~/tmp/node-elf
```

To use this directory as a Git repository, enter the following command (which will create a directory called “.git” that contains repository metadata):

```
git init
```

ADDING FILES TO A GIT REPOSITORY

Now that you’ve set up an empty repository, you’ll want to add some files. For this example, we’ll add a file containing the URL-shortening logic. Save the following listing’s 13.1 content into a file called “index.js” in this directory:

Listing 13.1 A Node module for URL shortening

```
exports.initPathData = function(pathData) { ①
  pathData = (pathData) ? pathData : {};
  pathData.count = (pathData.count) ? pathData.count : 0;
  pathData.map   = (pathData.map) ? pathData.map : {};
}
```

```

exports.shorten = function(pathData, path) { ②
  exports.initPathData(pathData);
  pathData.count++;
  pathData.map[pathData.count] = path;
  return pathData.count.toString(36);
}

exports.expand = function(pathData, shortened) { ③
  exports.initPathData(pathData);
  var pathIndex = parseInt(shortened, 36);
  return pathData.map[pathIndex];
}

```

- ① The initialization function is called implicitly by shorten() and expand()
- ② Accepts a "path" string and returns a shortened URL mapping to it
- ③ Accepts a previously shortened url and returns the expanded URL

Next, let Git know that you want this file in your repository. The git “add” command works differently than other version control systems. Instead of adding files to your repository, the git “add” command adds files to Git’s “staging area.” The staging area can be thought of as a checklist where you indicate newly added files, or files that you’ve changed that you’d like to be included in the next revision of your repository.

```
git add index.js
```

Git now knows that it should track this file. You could add other files to the “staging area” if you wanted to, but for now we only need to add this one file. To let Git know you’d like to make a new revision in the repository, including the changed files you’ve selected in the staging area, use the “commit” command. The commit command can, as with other version control tools, take a “-m” command-line flag to indicate a message describing the changes in the new revision.

```
git commit -m "Added URL shortening functionality."
```

The version of the repository on your workstation now contains a new revision. For a list of repository changes, enter the following command.

```
git log
```

PUSHING FROM GIT TO GITHUB

If your workstation was suddenly struck by lightning you'd lose all of your work. To safeguard against unexpected events, and get the full benefits of GitHub's web interface, you'll want to send changes you've made in your local Git repository to your GitHub account. But before doing this, you've got to let Git know where it should send changes to. To do this, you add a Git remote repository. These are referred to as "remotes." The following line shows how you add a GitHub remote to your repository. Substitute "username" with your username, and note that "node-elf" indicates the name of the project.

```
git remote add origin git@github.com:username/node-elf.git
```

Now that you've added a remote, you can send your changes to GitHub. In Git terminology sending changes is called a repository "push." In the following command, tell Git to push your work to the "origin" remote you defined earlier. Every Git repository can have one or more branches, which are, conceptually, separate working areas in the repository. You want to push your work into the "master" branch:

```
git push -u origin master
```

In the push command the "-u" option tells Git that this remote is the "upstream" remote and branch. The upstream remote is the default remote used. After doing your first push with the "-u" option, you'll be able to do future pushes by using the following command, which is easier to remember:

```
git push
```

If you go to GitHub and refresh your repository page you should now see your file.

Creating a module and hosting it on GitHub is a quick-and-dirty way to be able to reuse it. If you want, for example, to use your sample module in a project, you could enter the commands in the following example. The `require('elf')`

would then provide access to the module. Note that when cloning the repository, you use the last command-line argument to name the directory into which you’re cloning.

```
mkdir ~/tmp/my_project/node_modules
cd ~/tmp/my_project/node_modules
git clone https://github.com/mcantelon/node-elf.git elf
cd ..
```

You now know how to add projects to GitHub, including how to create a repository on GitHub, how to create and add files to a Git repository on your workstation, and how to push your workstation repository to GitHub. Given the open source world’s embrace of Git, you’ll find many excellent resources to help you go further. If you’re looking for comprehensive instruction on how to use Git, Scott Chacon, one of the founders of GitHub, has written a thorough book that you can purchase or read free online⁸. If a hands-on approach is more your style, the official Git site’s documentation page⁹ lists a number of tutorials that will get you up and running.

Footnote 8 <http://progit.org/>

Footnote 9 <http://git-scm.com/documentation>

13.2.3 Collaborating using GitHub

Now that you know how to create a GitHub repository from scratch, we’ll look at how you can use GitHub to collaborate with others.

Let’s say you’re using a third-party module and run into a bug. You may be able to examine the module’s source code and figure out a way to fix it. If you do, you could email the author of the code, describing your fix and attaching files containing your fixes. But this would require the author to do some tedious work. The author would have to compare your files to her own and combine the fixes from your files with the latest code. But if the author was using GitHub, you could make a “clone” of the author’s project repository, make some changes, then inform the author via GitHub of the bug fix. GitHub would then show the author, on a web page, the differences between your code and the version you duplicated and, if the bug fix is acceptable, combine the fixes with the latest code via a single mouse click.

In GitHub parlance, duplicating a repository is known as “forking.” Forking a project allows you to do anything you want to your copy with no danger to the

original repository. You don't need the permission of the original author to fork: anyone can fork any project and submit their contributions back to the original project. The original author may not approve your contribution, but if not you still have your own fixed version, which you can continue to maintain and enhance independently. If your fork were to grow in popularity, others might well fork your fork, and offer contributions of their own.

Once you've made changes to a fork, submitting these changes to the original author is called a "pull request." A pull request is a message asking a repository author to "pull" changes. Pulling, in Git parlance, means importing work from a fork and combining the work with your own. Figure 13.5 shows, visually, an example GitHub collaboration scenario.

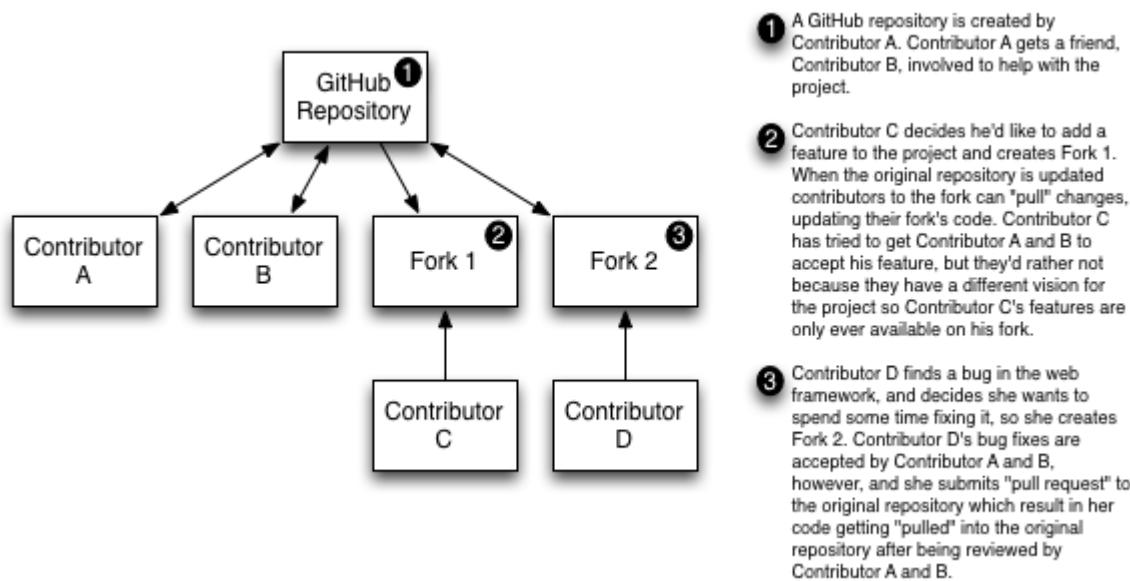


Figure 13.5 An example of a typical GitHub development scenario.

Now, let's walk through an example, represented visually in figure 13.6, of forking a GitHub repository for the purpose of collaboration.

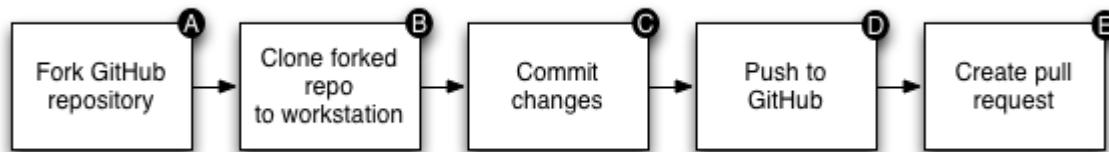


Figure 13.6 The process of collaborating on GitHub via forking.

Forking starts the collaboration process by duplicating the repository on GitHub to your own account (known as "forking") [A]. You then clone the forked

repository to your workstation [B], make changes to it, commit the changes [C], push your work back to GitHub [D], and send a pull request to the owner of the original repository asking them to consider your changes [E]. If they want to include your changes in their repository they will approve your pull request.

Let's say you want to fork the `node-elf` repository you created earlier in this chapter and add it to the URL-shortening module, which would specify the module's version number by the module logic itself. This would allow anyone using the module to ensure they're using the right version and not some other version by accident.

First, log into GitHub then navigate to the repository's main page: <https://github.com/mcantelon/node-elf>. Once at the repository page you'd click "Fork" to duplicate the repository. The resulting page is similar to the original repository page with "forked from mcantelon/node-elf" displayed under the repository name.

After forking, the next step is to clone the repository to your workstation, make your changes, and push the changes to GitHub. Using this example, the following commands will do the same using the "`node-elf`" repository:

```
mkdir -p ~/tmp/forktest
cd ~/tmp/forktest
git clone git@github.com:chickentown/node-elf.git
cd node-elf
echo "exports.version = '0.0.2';" >> index.js
git add index.js
git commit -m "Added specification of module version."
git push origin master
```

Once you've pushed your changes, click "Pull Request" on your fork's repository page and enter the subject and body of a message describing your changes. Click "Send pull request." Figure 13.7 shows a screenshot containing typical content:

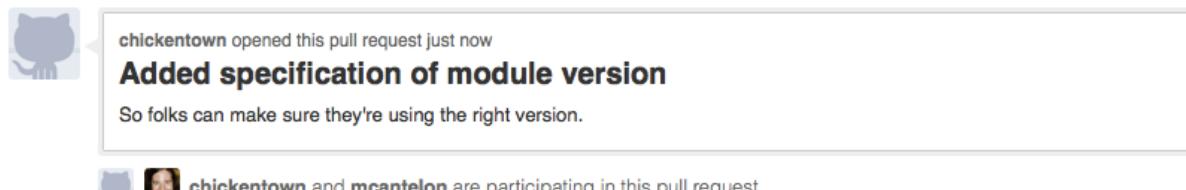


Figure 13.7 The details of a GitHub pull request.

The pull request then gets added to the issue queue of the original repository.

The owner of the original repository can then, after reviewing your changes, incorporate them by clicking “Merge pull request,” entering a commit message, and clicking “Confirm Merge.” This automatically closes the issue.

Once you’ve collaborated with someone and created a great module, the next step is getting it out to the world. The best way is to add it to the npm repository. Next, you’ll learn how to publish to npm.

13.3 Contributing to the Node Package Manager repository

Suppose you’ve worked on the URL-shortening module for some time and think it would be useful to other Node users. To publicize it, you could post on Node-related Google Groups talking about its functionality. But you’d be limited in the number of Node users you’d reach, and as people start using your module you wouldn’t have a way to let them know about updates to your module.

To solve the problems of discoverability and providing updates you can publish to npm. With npm you can easily define a project’s dependencies allowing them to be automatically installed at the same time as your module. If you’ve created a module designed to store comments about content (blog posts, for example), you could have a module handling MongoDB storage of comment data included as a dependency. Another example would be a module that provides a command-line tool that might have as a dependency a helper module for parsing command-line arguments.

Up to this point in the book, you’ve used npm to install everything from testing frameworks to database drivers, but you haven’t yet published anything. In the next section we’ll show you each step you need to take to publish your own work on npm:

- Preparing a package
- Writing a package specification
- Testing a package
- Publishing a package

13.3.1 Preparing a package

Any Node module you want to share with the world should be accompanied by related resources such as documentation, examples, tests, and related command-line utilities. The module should come with a “readme” file that provides enough information to get you started quickly.

The package directory should be organized using subdirectories. Table 13.2 lists conventional subdirectories “bin,” “docs,” “example,” “lib,” and “test,” and for what you’d use each of them.

Table 13.2 Conventional subdirectories in a Node project

Directory	Use
bin	Command-line scripts
docs	Documentation
example	Example application uses
lib	Core application functionality
test	Test scripts and related resources

Once you’ve organized your package, you’ll want to prepare it for publishing to npm by writing a package specification.

13.3.2 Writing a package specification

When you publish a package to npm, you need to include a machine-readable package specification file. This JSON file is called “package.json” and includes information about your module, such as its name, description, version, dependencies, and other characteristics. Nodejitsu has a handy website¹⁰ that shows a sample package.json file and explains what each part of the sample file is for when you hover your mouse over it.

Footnote 10 <http://package.json.nodejitsu.com/>

In a package.json file, only the name and version are mandatory. Other characteristics are optional, but some, if defined, can make your module more useful. By defining a “bin” characteristic, for example, you can let npm know which files in your package are meant to be command-line tools and npm will make them globally available.

The following shows what a sample specification might look like:

```
{
  "name": "elf",
  "version": "0.0.1",
  "description": "Toy URL shortener",
  "author": "Mike Cantelon <mcantelon@example.com>",
  "main": "index",
  "engines": { "node": "0.4.x" }
}
```

For comprehensive documentation on available package.json options, enter the following:

```
npm help json
```

Because generating JSON by hand is only slightly more fun than hand-coding XML, let's look at some tools that make it easier. One such tool, ngen, is an npm package that when installed, adds a command-line tool called “ngen.” After asking a number of questions, ngen will generate a package.json file. It'll also generate a number of other files that are normally included in npm packages, such as a “Readme.md” file.

You can install ngen using the following command line:

```
npm install -g ngen
```

After you've run ngen in the root directory of a module and you're ready to publish to npm, you'll end up with output like you see in the following code. Some files may be generated that you don't need. Delete them. A .gitignore file will be added, for example, that specifies a number of files and directories that shouldn't normally be added to the Git repository of a project that will be published to npm. A .npmignore file will also be added, which serves a similar function, letting npm know what files can be ignored when publishing the package to npm.

```
Project name: elf
Enter your name: Mike Cantelon
Enter your email: mcantelon@gmail.com
Project description: URL shortening library
create : /Users/mike/programming/js/shorten/node_modules/.gitignore
```

```
create : /Users/mike/programming/js/shorten/node_modules/.npmignore
create : /Users/mike/programming/js/shorten/node_modules/History.md
create : /Users/mike/programming/js/shorten/node_modules/index.js
...
```

Generating a package.json file is the hardest part of publishing to npm. Now that you've taken this step, you're ready to publish your module.

13.3.3 Testing and publishing a package

Publishing a module to npm involves three steps which we will go over in this section:

- Testing the installation of your package locally
- Adding an npm user, if you haven't already
- Publishing the package to npm

TESTING PACKAGE INSTALLATION

To test a package locally, use npm's "link" command from the root directory of your module. This command makes your package globally available on your workstation, where Node can use it, similar to a package conventionally installed by npm.

```
sudo npm link
```

Next, create a test directory in which you can link-install the package to test it:

```
npm link elf
```

When you've link installed the package, do a quick test of requiring the module by executing the require function in the Node REPL, as the following code shows. In the results you should see the variables or functions that your module provides.

```
node
> require('elf');
{ version: '0.0.1',
  initPathData: [Function],
  shorten: [Function],
  expand: [Function] }
```

If your package passed the test and you've finished developing it, use npm's "unlink" command from the root directory of your module. Your module will no longer be globally available on your workstation, but later, once you've completed publishing your module to npm, you'll be able to install it normally using the "install" command.

```
sudo npm unlink
```

Having tested your npm package, the next step is to create an npm publishing account, if you haven't previously set one up.

ADDING AN NPM USER

Enter the following to create your own npm publishing account:

```
npm adduser
```

You'll be prompted for a username, an email address, and a password. If your account is successfully added, you won't see an error message.

PUBLISHING TO NPM

The next step is to publish. Enter the following to publish your package to npm:

```
npm publish
```

You may see the warning "Sending authorization over an insecure channel" but if you don't see additional errors, your module was published successfully. You can verify your publish was successful by using npm's "view" command:

```
npm view elf description
```

If you'd like to include one or more private repositories as npm package dependencies, you can. Perhaps you have a module of useful helper functions you'd like to use, but not release publicly on npm. To add a private dependency, where you would normally put the dependency module's name, you can put any name that's different than the other dependency names. Where you would normally put the version, you put a Git repository URL. In the following example, an excerpt from a package.json file, the last dependency is a private repository:

```
"dependencies" : {
  "optimist" : ">=0.1.3",
  "iniparser" : ">=1.0.1",
  "mingy": ">=0.1.2",
  "elf": "git://github.com/mcantelon/node-elf.git"
},
```

Note that any private modules should also include package.json files. To make sure you don't accidentally publish one of these modules, set the "private" property in its package.json file to "true":

```
"private": true,
```

Now that you know how to get help, collaborate, and contribute you're ready to dive right into Node community waters.

13.4 Summary

As with most successful open source projects, Node has an active online community, which means you'll find plenty of available online resources as well as quick answers to your questions using online references, Google Groups, IRC, or GitHub issue queues.

In addition to a place where projects keep track of bugs, GitHub also provides Git hosting and the ability to browse Git repository code using a web browser. Using GitHub, other developers can easily "fork" your open source code if they want to contribute bug fixes, add features, or take a project in a new direction. You can also easily submit changes made to a fork back to the original repository.

Once a Node project has reached the stage where it's worth sharing with the world, you can submit it to the Node Package Manager repository. Inclusion in npm makes your project easier for others to find, and if your project is a module, inclusion in npm means your module will be easy to install.

You know how to get the help you need, collaborate online, and share your work. Now it's time to get the most out of Node development.

Installing Node and community add-ons



Node is easy to install on most operating systems. Node can either be installed using conventional application installers or by using the command-line. Command-line installation is easy on OS X and Linux, but not recommended for Windows.

To help you get started, we've detailed the Node installation on OS X, Windows, and Linux operating systems. We've also explained how you can use the Node Package manager (npm) to find and install useful add-ons.

A.1 OS X setup

Installing Node on OS X is quite straightforward. The official installer¹, shown in figure A.1, provides an easy way to install a precompiled version of Node and npm.

Footnote 1 <http://nodejs.org/#download>

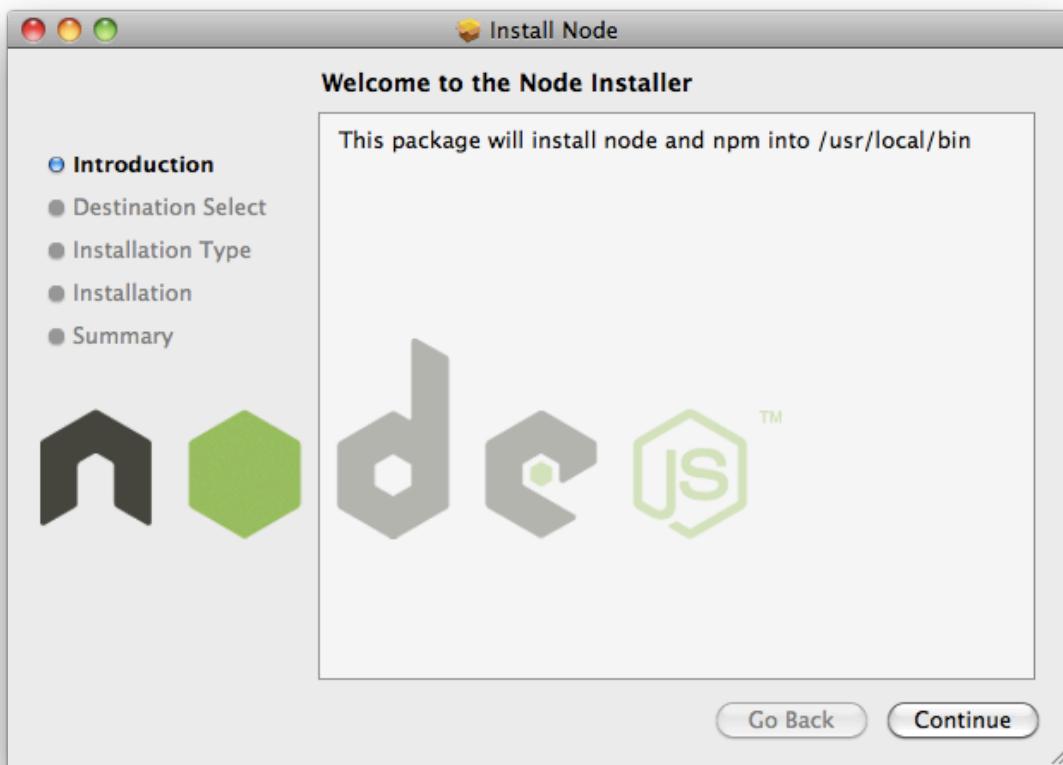


Figure A.1 The official Node installer for OS X

If you'd rather install from source, however, you can either use a tool called Homebrew², which automates installation from source, or you can manually install from source. Installing Node from source on OS X, however, requires you to have XCode developer tools installed.

Footnote 2 <http://mxcl.github.com/homebrew/>

NOTE**XCode**

If you don't have XCode installed, you can download XCode from Apple's website³. You'll have to register with Apple as a developer, which is free, to access the download page. The full XCode installation is a large download (approximately 4GB), so as an alternative Apple also offers Command Line Tools for Xcode which is available for download on the same web page and gives you the minimal functionality needed to compile Node and other open source software projects.

Footnote 3 <http://developer.apple.com/downloads/>

To quickly check if you have XCode, you can start the Terminal application and run the command `xcodebuild`. If you have XCode installed you should get an error indicating that your current directory "does not contain an Xcode project".

Either method requires entering OS X's command-line interface by running the Terminal application that is usually found in the "Utilities" folder in the main "Applications" folder.

If compiling from source, see A.4 "Compiling Node", later in this section, for the necessary steps.

A.1.1 Installation with Homebrew

An easy way to install Node on OS X is by using Homebrew, an application for managing the installation of open source software.

Install Homebrew by entering the following into the command-line:

```
ruby -e "$(curl -fsSkL raw.githubusercontent.com/mxcl/homebrew/go)"
```

Once Homebrew is installed you can install Node by entering the following:

```
brew install node
```

As Homebrew compiles code you'll see a lot of text scroll by. The text is information related to the compiling process and can be ignored.

A.2 Windows setup

Node can be most easily installed on Windows by using the official stand-alone installer⁴. After installing you'll be able to run Node and npm from the Windows command-line.

Footnote 4 <http://nodejs.org/#download>

An alternative way to install Node on Windows involves compiling it from source code. This is more complicated and requires the use of a project called Cygwin that provides a Unix compatible environment. You'll likely want to avoid using Node through Cygwin unless you're trying to use modules that won't otherwise work on Windows or need to be compiled, such as some database driver modules.

To install Cygwin, navigate to the Cygwin installer download link⁵ in your web browser and download setup.exe. Double-click setup.exe to start installation then click "Next" repeatedly to select the default options until you reach the "Choose a Download Site" step. Select any of the download sites from the list and click "Next". If you see a warning pop-up about Cygwin being a major release, click "OK" to continue.

Footnote 5 <http://www.cygwin.com/setup.exe>

You should now see Cygwin's package selector, as shown in figure A.2.

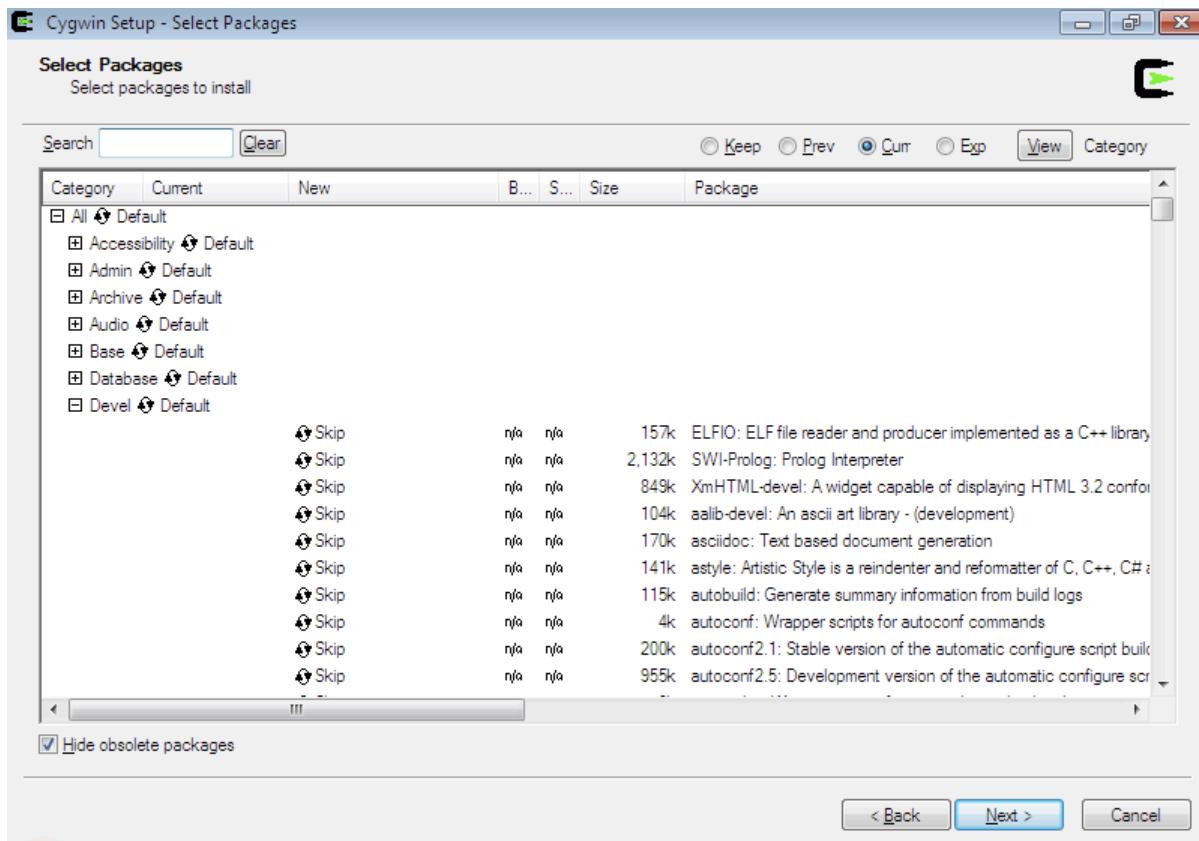


Figure A.2 The Cygwin's package selector allows you to select open source software that will be installed on your system.

You'll use this selector to pick what software functionality you'd like installed in your Unix-like environment (see table A.1 for a list of Node development-related packages to install).

Table A.1 Cygwin packages needed to run Node

Category	Package
devel	gcc4-g++
devel	git
devel	make
devel	openssl-devel
devel	pkg-config
devel	zlib-devel
net	inetutils
python	python
web	wget

Once you've selected the packages click "Next".

You'll then see a list of packages that the ones you've selected depend on. You need to install those as well, so click "Next" again to accept them. Cygwin will now download the packages you need. Once the download has completed, click "Finish".

Start Cygwin by clicking the desktop icon or start menu item. You'll be presented with a command-line prompt. You then compile Node (see A.4 for the necessary steps).

A.3 Linux setup

Installing Node on Linux is usually painless. We'll run through installations, from source code, on two popular Linux distributions: Ubuntu and CentOS. Node is also available through package managers on a number of distributions⁶.

Footnote 6 <https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager>

A.3.1 Ubuntu installaton prerequisites

Before installing Node on Ubuntu, you'll need to install prerequisite packages. This is done on Ubuntu 11.04 or later using a single command:

```
sudo apt-get install build-essential libssl-dev
```

NOTE

Sudo

The "sudo" command is used to perform another command as "superuser" (also referred to as "root"). Sudo is often used during software installation because files needs to be placed in protected areas of the filesystem and the superuser can access any file on the system regardless of file permissions.

A.3.2 CentOS installaton prerequisites

Before installing Node on CentOS, you'll need to install prerequisite packages. This is done on CentOS 5 using the following commands:

```
sudo yum groupinstall 'Development Tools'
sudo yum install openssl-devel
```

Now that you've installed the prerequisites you can move on to compiling Node.

A.4 Compiling Node

Compiling Node involves the same steps on all operating systems.

In the command-line, you first enter the following command to create a temporary folder in which to download the Node source code:

```
mkdir tmp
```

Next you navigate into the directory created in the previous step:

```
cd tmp
```

You now enter the following command:

```
curl -O http://nodejs.org/dist/node-latest.tar.gz
```

Next, you'll see text indicating the download progress. Once progress reaches 100% you're returned to the command prompt. Enter the following command to decompress the file you received:

```
tar zxvf node-latest.tar.gz
```

You should then see a lot of output scroll past be returned to the command prompt. Once returned to the prompt, enter the following to list the files in the current folder, which should include the name of the directory you just decompressed:

```
ls
```

Next, enter the following command to move into this directory:

```
cd node-v*
```

You are now in the directory containing Node's source code. Enter the

following to run a configuration script that will prepare the right installation for your specific system:

```
./configure
```

Next, enter the following to compile Node:

```
make
```

NOTE

Cygwin Quirk

If you're running Cygwin on Windows 7 or Vista you may run into errors during this step. These are due to an issue with Cygwin rather than an issue with Node. To address them, exit the Cygwin shell then run the ash.exe command-line application (located in the Cygwin directory: usually c:\cygwin\bin\ash.exe). In the ash command-line enter "/bin/rebaseall -v". When this completes, restart your computer. This should fix your Cygwin issues.

Node normally takes a little while to compile, so be patient and expect to see a lot of text scroll by. The text is information related to the compiling process and can be ignored.

At this point, you're almost done! Once text stops scrolling and you again see the command prompt you can enter the final command in the installation process:

```
sudo make install
```

When finished, enter the following to run Node and have it display its version number, verifying that it has been successfully installed:

```
node -v
```

You should now have Node on your machine!

A.5 Using the Node Package Manager

With Node installed you'll be able to use built-in modules that provide you with APIs to perform networking tasks, interact with the file system, and do other things commonly needed in applications. Node's built-in modules are referred to collectively as the Node "core". While Node's core encompasses a lot of useful functionality, you'll likely want to use community-created functionality as well. Figure A.3 shows, conceptually, the relationship between the Node core and add-on modules.

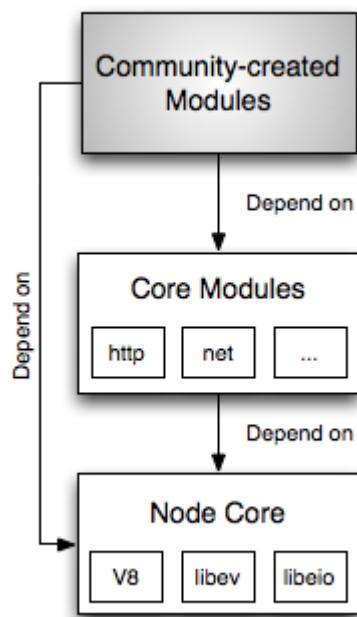


Figure A.3 The Node stack is composed of globally available functionality, core modules, and community-created modules.

Depending on what language you've been working in, you may or may not be familiar with the idea of community repositories of add-on functionality. These repositories are akin to a library of useful application building blocks that can help you do things that the language itself doesn't easily allow out-of-the-box. These repositories are usually modular: rather than fetching the entire library all at once, you can usually fetch just the libraries you need.

The Node community has its own tool for managing community add-ons: the Node Package Manager (npm). In this section you'll learn how to use npm to find

community add-ons, view add-on documentation, and explore the source code of add-ons.

SIDE BAR **npm is missing on my system**

If you've installed Node, then npm is likely already installed. You can test by running `npm` on the command-line and see if the command is found. If not, you can install npm by doing the following:

```
cd /tmp
git clone git://github.com/isaacs/npm.git
cd npm
sudo make install
```

Once you've installed npm, enter the following on a command-line to verify npm is working (by asking it to output its version number):

```
npm -v
```

If npm has installed correctly, you should see a number similar to the following:

```
1.0.3
```

If you do, however, run into problems installing npm, the best thing to do is to visit the npm project on Github⁷ where the latest installation instructions can be found.

Footnote 7 <http://github.com/isaacs/npm>

A.5.1 Searching for packages

The npm command-line tool provides convenient access to community add-ons. These add-on modules are referred to as "packages" and are stored in an online repository. For users of PHP, Ruby, and Perl npm is analogous to Pear, Gem, and CPAN respectively.

The npm tool is extremely convenient. With npm you can download and install

a package using a single command. You can also easily search for packages, view package documentation, explore a package's source code, and publish your own packages so they can be shared with the Node community.

You can use npm's `search` command to find packages available in its repository. For example, if you wanted to search for an XML generator, you could simply enter the command:

```
npm search xml generator
```

The first time npm does a search, there's a long pause as it downloads repository information. Subsequent searches, however, are quick.

As an alternative to command-line searching, the npm project also maintains a web search interface⁸ to the repository. This website, shown in figure A.4, also provides statistics on how many packages exist, which packages are the most depended on by others, and which packages have recently been updated.

Footnote 8 <http://search.npmjs.org/>

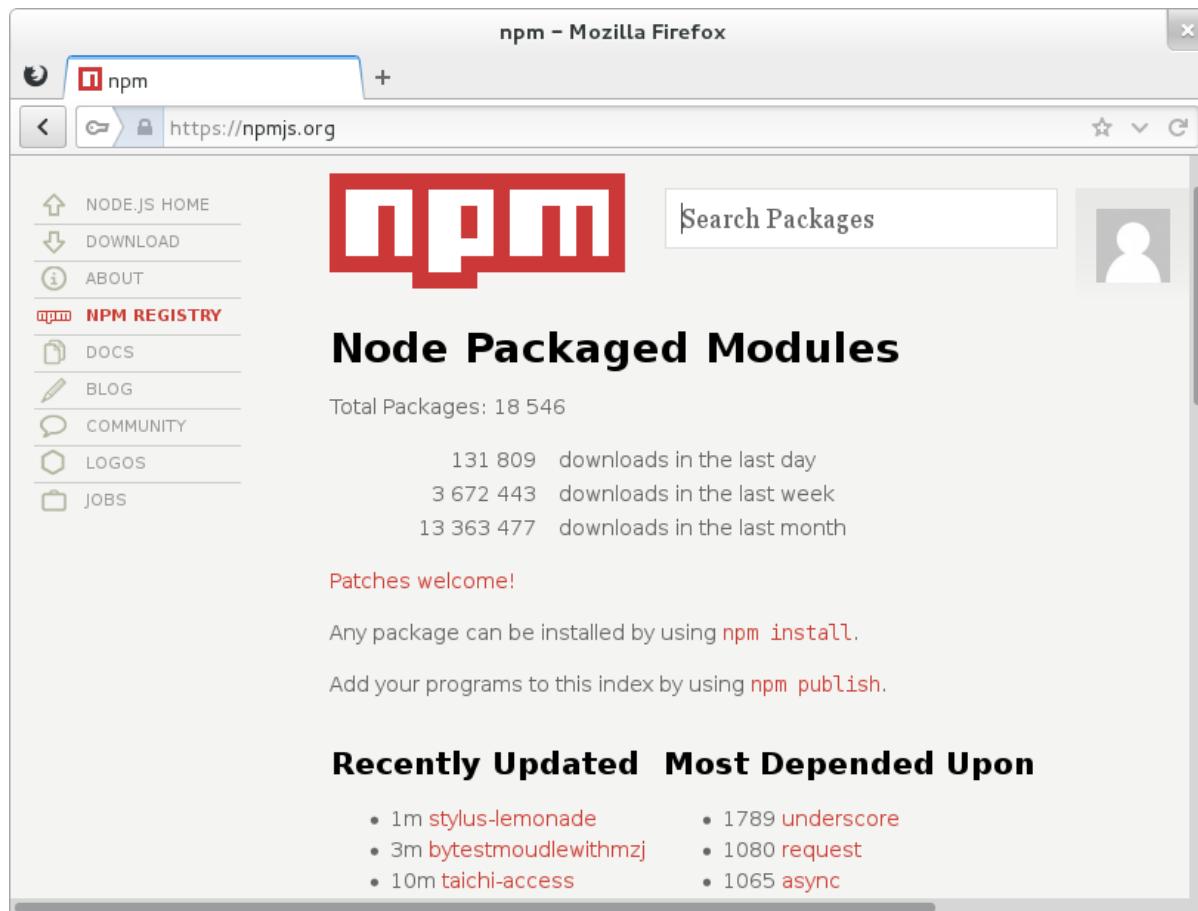


Figure A.4 The npm search website provides useful statistics on module popularity.

The npm web search interface also lets you browse individual packages, which shows useful data such as the package dependencies and the online location of a project's version control repository.

A.5.2 Installing packages

Once you've found packages you'd like to install, there are two main ways of doing so using npm: locally and globally.

Locally installing a package puts the downloaded module into folder called "node_modules" in the current working directory. If this folder doesn't exist, npm will create it.

Here's an example of installing the "express" package locally:

```
npm install express
```

Globally installing a package puts the downloaded module into the "/usr/local" directory on non-Windows operating systems, a directory traditionally used by

Unix to store user installed applications. In Windows, the "Appdata\Roaming\npm" subdirectory of your user directory is where globally installed npm modules are put.

Here's an example of installing the "express" package globally:

```
npm install -g express
```

If you don't have sufficient file permissions when installing globally you may have to prefix your command with sudo. For example:

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

After you've installed a package, the next step is figuring out how it works. Luckily, npm makes this easy.

A.5.3 Exploring Documentation and Package Code

The npm tool offers a convenient way to view a package author's online documentation, when available. The "docs" npm command will open a web browser with a specified package's documentation. Here's an example of viewing documentation for the "express" package:

```
npm docs express
```

You can view package documentation even if the package isn't installed.

If a package's documentation is incomplete or unclear, it's often handy to be able to check out the package's source files. The npm tool provides an easy way to spawn a subshell with the working directory set to the top-level directory of a package's source files. Here's an example of exploring the source files of a locally installed "express" package:

```
npm explore express
```

To explore the source of a globally installed package, simply add the "-g" command-line option after "npm". For example:

```
npm -g explore express
```

Exploring a package is also a great way to learn. Reading Node source code often introduces you to unfamiliar programming techniques and ways of organizing code.

B *Debugging Node*

During development, and especially while learning a new language or framework, debugging tools and techniques can be helpful. In this appendix you'll learn a number of ways to figure out exactly what's going on with your Node application code.

B.1 Analyzing code with JSHint

Syntax and scope-related errors are a common pitfall of development. The first line of defense, when attempting to determine the root of an application problem, is to look at the code. If you look at the source code, however, and don't immediately see a problem another thing worth doing is running a utility to check your source code for problems.

JSHint is one such utility. This utility can alert you to show-stopping errors, such as functions called in code that aren't defined anywhere, as well as stylistic concerns, such as not heeding the JavaScript convention of capitalizing class constructors. Even if you never run JSHint, reading over the types of errors it looks for will alert you to possible coding pitfalls.

JSHint is a project based on JSLint, a JavaScript source code analysis tool that has been around for a decade. JSLint, however, is not very configurable and that's where JSHint comes in.

JSLint is, in the opinion of many, overly strict in terms of enforcing stylistic recommendations. JSHint, conversely, allows you to tell it what you want it to check for and what you want it to ignore. Semi-colons, for example, are technically required by JavaScript interpreters, but most interpreters use automated semi-colon insertion (ASI) to insert them where they're missing. Because of this, some developers omit them in their source code to lessen visual noise and their code runs

without issue. While JSLint would complain that a lack of semi-colons is an error, JSHint can be configured to ignore this "error" and check for show-stopping issues.

Installing JSHint makes available a command line tool called `jshint` that checks source code. JSHint should be installed globally using npm by executing the following command.

```
npm install -g jshint
```

Once you've installed JSHint, you can check JavaScript files by simply entering something like the following example.

```
jshint my_app.js
```

You'll most likely want to create a configuration file for JSHint that indicates what you want it to check. One way to do so is to copy the default configuration file, available on GitHub¹, to your workstation and modify it.

Footnote 1 <https://github.com/jshint/node-jshint/blob/master/.jshintrc>

If you name your version of the config file `.jshintrc` and include it in your application directory, or in any parent directory of your application directory, JSHint will automatically find and use it.

Alternatively, you can run JSHint using the `config` flag to specify a configuration file location. The following example shows JSHint being told to use a configuration file with a non-standard filename.

```
jshint my_app.js --config /home/mike/jshint.json
```

For details about each specific configuration option, check out the JSHint website².

Footnote 2 <http://www.jshint.com/options/>

B.2 Outputting debugging information

If your code appears to be legitimate, but your application is still behaving unexpectedly, you may want to add debugging output to get a better sense of what's going on underneath the hood.

B.2.1 Debugging with the `console` module

The `console` module is a built-in Node module that provides functionality useful for console output and debugging.

OUTPUTTING APPLICATION STATUS INFORMATION

The `console.log` function is used to output application status information to standard output. The `console.info` is another name for the same function. Arguments can be provided, `printf()` style³, as the following example shows.

Footnote 3 <http://en.wikipedia.org/wiki/Printf>

```
console.log('Counter: %d', counter);
```

For outputting warnings and errors, the `console.warn` and `console.error` functions operate similarly. The only difference is that instead of printing to standard output they print to standard error. This enables you to, if desired, redirect warnings and errors to a log file as the example below shows.

```
node server.js 2> error.log
```

When outputting the contents of an object, the `console.dir` function will output an object's contents. The following example output shows what this looks like.

```
{ name: 'Paul Robeson',
  interests: [ 'football', 'politics', 'music', 'acting' ] }
```

OUTPUTTING TIMING INFORMATION

The `console` module includes two functions that, when used together, allow you to time the execution of parts of your code. More than one thing can be timed simultaneously.

To start timing, add the following line to your code at the point you'd like timing to start.

```
console.time('myComponent');
```

To end timing, returning the time elapsed since timing started, add the following line to your code at the point where timing should stop.

```
console.timeEnd('myComponent');
```

The above line will display the elapsed time.

OUTPUTTING STACK TRACES

A stack trace provides you with information about what functions executed before a certain point in application logic. When Node encounters an error during application execution, for example, it outputs a stack trace to provide information about what led, in the application's logic, to the error.

At any point during your application you can output a stack trace, without causing your application to stop, by executing `console.trace()`.

This will produce output similar to the following example stack trace.

```
Trace:
  at lastFunction (/Users/mike/tmp/app.js:12:11)
  at secondFunction (/Users/mike/tmp/app.js:8:3)
  at firstFunction (/Users/mike/tmp/app.js:4:3)
  at Object.<anonymous> (/Users/mike/tmp/app.js:15:3)
  ...
...
```

Note that stack traces display execution in reverse chronological order.

B.2.2 Using the debug module to manage debugging output

While debugging output is useful, if you're not actively troubleshooting an issue debugging output can end up just being visual noise. Ideally you could switch debugging output on or off.

One technique to toggle debugging output is through the use of an environmental variable. TJ Holowaychuk's debug module provides a handy tool for this, allowing you to manage debugging output using the DEBUG environmental variable. Chapter 12 details the use of the debug module.

B.3 Node's built-in debugger

For debugging needs beyond adding simple debugging output, Node comes with a built-in command-line debugger. The debugger is invoked by starting your application using the "debug" keyword, as the following example shows.

```
node debug server.js
```

When running a Node application this way, you'll be shown the first few lines of your application and presented with a debugger prompt, as shown in figure B.1.

```
$ node debug server.js
< debugger listening on port 5858
connecting... ok
break in server.js:1
1 var http = require('http');
2
3 http.createServer(function (req, res) {
debug> |
```

Figure B.1 Starting the build-in Node debugger

The "break in server.js:1" means the debugger has stopped before executing the first line.

B.3.1 Debugger navigation

On the debugger prompt you can control the execution of your application. You could enter `next` (or just `n`) to execute the next line or, alternatively, you could enter `cont` (or just `c`) to have it execute until interrupted.

The debugger can be interrupted by the termination of the application or by what is called a "breakpoint". Breakpoints are points where you wish the debugger to stop execution so you can examine application state. One way to add a breakpoint is by adding a line to your application where you wish to put the breakpoint. This line should contain the statement `debugger;` as listing B.1 shows. The `debugger;` line won't do anything while running Node normally so you can leave it and there will be no ill effects.

Listing B.1 breakpoint.js: Adding a breakpoint programmatically

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

function handleRequest(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}

http.createServer(function (req, res) {
  debugger;           ① Added a breakpoint  
to code
  handleRequest(req, res);
}).listen(1337, '127.0.0.1');

console.log('Server running at http://127.0.0.1:1337/');
```

When running listing B.1 in debug mode it will first break at line one. If you now enter `cont` into the debugger it will proceed to create the HTTP server, awaiting a connection. If you create a connection by visiting `http://127.0.0.1:1337` with a browser then you'll see that it breaks at the `debugger;` line.

Enter `next` to continue to the next line of code. The current line will now be a function call to `handleRequest`. If you again enter `next` to continue to the next line the debugger won't descend into each line of `handleRequest`. If you, however, enter `step` then the debugger will descend into the `handleRequest`

function, allowing you to troubleshoot any issues with this particular function. If you changed your mind about wanting to debug `handleRequest` you can then enter `out` (or `o`) to "step out" of the function.

Breakpoints can be set from within the debugger in addition to being specified in source code. To put a breakpoint in the current line in the debugger enter `setBreakpoint()` (or `sb()`) into the debugger. It is possible to set a breakpoint at a specific line (`sb(line)`) or when a specific function is being executed (`sb('fn()')`).

For the times that you want to unset a breakpoint, there's the `clearBreakpoint()` function (`cb()` for short). This function takes the same arguments as the `setBreakpoint()` function, only it does the inverse.

B.3.2 Examining and manipulating state in the debugger

If you want to keep an eye on particular values in the application, you can add what are called "watchers". Watchers inform you of the value of a variable as you navigate through code.

For example, when debugging the code in listing B.1 you could enter `watch("req.headers['user-agent']")` and, for each step, you'd see what type of browser made the request. To see a list of watchers you'd enter the `watchers` command. To remove a watch you'd use the `unwatch` command, `unwatch("req.headers['user-agent'])` for example.

If at any point during debugging, you want to be able to fully examine, or manipulate, the state you can use the `repl` command. This allows you to enter any JavaScript expression and have it evaluate. To exit the REPL and return to the debugger press `CTRL+C`.

Once you're done debugging, you can exit the debugger by pressing `CTRL-C` twice, pressing `CTRL-D`, or by entering the `.exit` command.

These are the basics of debugger use. For more information on what can be done with the debugger, visit the official Node page⁴ for it.

Footnote 4 <http://nodejs.org/api/debugger.html>

B.4 Node Inspector

Node Inspector is an alternative to Node's built-in debugger that uses a WebKit-based browser such as Chrome or Safari, rather than the command-line, as an interface.

B.4.1 Starting Node Inspector

Before you begin debugging, Node Inspector should be installed globally with the following npm command. After installation, the `node-inspector` command will be available on your system.

```
npm install -g node-inspector
```

To debug a Node application, start it using the `--debug-brk` command-line option, as shown in the following example.

```
node --debug-brk server.js
```

Using the `--debug-brk` option causes the debugging to insert a breakpoint before the first line of your application. If this isn't desired, you can use the `--debug` option instead.

Once your application is running, start Node Inspector by entering the following command.

```
node-inspector
```

Node Inspector is interesting because it actually uses the same code as WebKit's Web Inspector, just plugged into Node's JS engine instead, so web developers should feel right at home using it. Once Node Inspector is running, navigate to `http://127.0.0.1:8080/debug?port=5858` in your WebKit browser and you should see the Web Inspector. If you've ran Node Inspector using the `--debug-brk` option it will immediately show the first script in your application, similar to Figure B.2. If you've used the `--debug` option you'll have to use the script selector, indicated by the script name "step.js" in figure B.2, to select the script you'd like to debug.

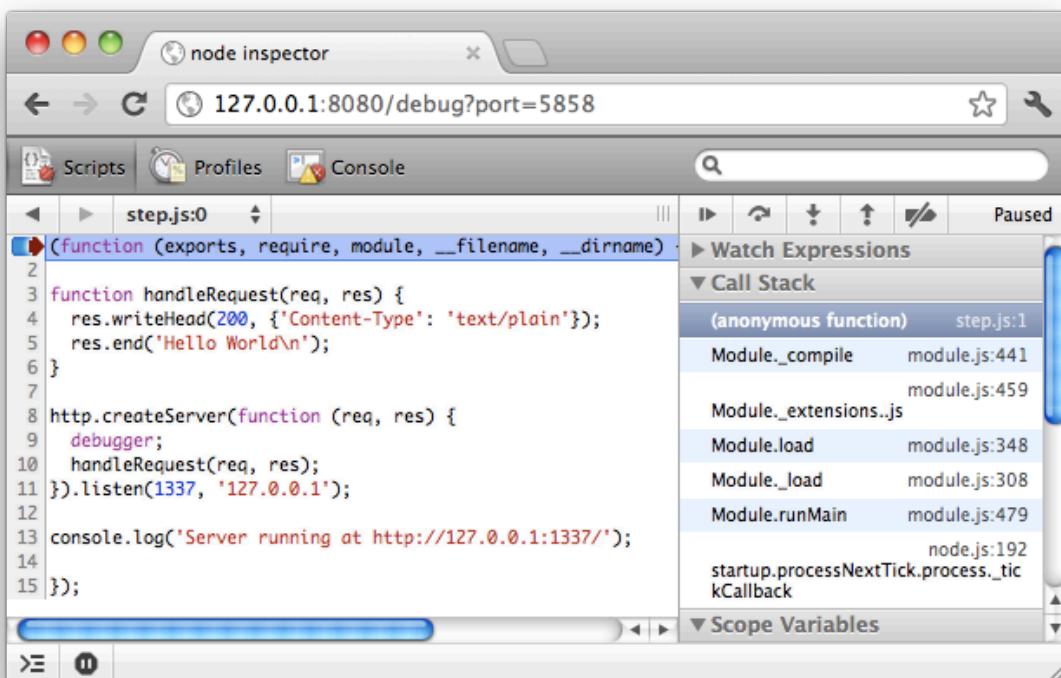


Figure B.2 The Web Inspector

A red arrow is shown to the left of the line of code that will execute next.

B.4.2 Node Inspector navigation

To step to the next function call in your application, click the button that looks like a small circle with an arc over it. Node Inspector, like the command-line Node debugger, allows you to step into functions as well. When the red arrow is to the left of a function call you can descend into the function by clicking the button that looks like a small circle with an arrow pointing down at it. To step out of the function click the button that looks like a small circle with an arrow pointing up. If you're using Node core or community modules the debugger will switch to script files for these modules as you step through your application. Don't be alarmed: it will at some point return to your application's code.

To add breakpoints while running Node Inspector, click the line number to the left of any line of a script. If you'd like to clear all breakpoints, click the button to the right of the step out button.

Node Inspector also has the interesting feature of allowing you to change code as your application runs. If you'd like to change a line of code, simply double click on it, edit it, then click out of the line.

B.4.3 Browsing state in Node Inspector

As you debug your application, you can inspect state using the collapsible panes under the buttons that allow you to navigate in the debugger, as shown in figure B.3. These allow you to inspect the call stack and variables in the scope of code currently being executed. Variables can be manipulated by double-clicking on them and changing their values. You can also, as with Node's built-in command-line debugger, add watch expressions that will display as you step through your application.

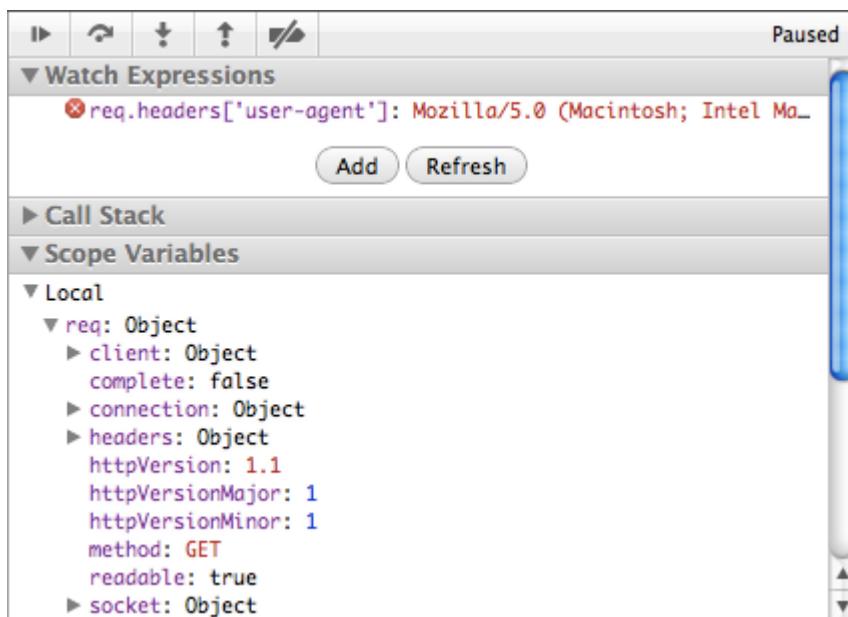


Figure B.3 Browsing application state using Node Inspector

For more details about how to get the most out of Node Inspector, visit its GitHub project page⁵.

Footnote 5 <https://github.com/dannycoates/node-inspector/>

TIP

When in doubt, refresh

While using Node Inspector and running into any odd behavior, refreshing the browser may help. If that doesn't work, try restarting both your Node application and Node Inspector.