Node.js IN ACTION

Mike Cantelon
TJ Holowaychuk
Nathan Rajlich

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Node.js in Action version 8

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

Table of contents

1 Why the Web needs Node

Imagine if web apps could change in real time to provide information instantly: if you could see a friend’s email as she types it, if you could see the bus approaching on your Smartphone web browser’s map, if you could play large multiplayer arcade games using a web browser without special plugins.

The Web has, somewhat, reached that level of capability, and becomes richer and faster as it continues to evolve. Advances in browser technology (such as HTML, CSS3, and WebGL) hint at unprecedented levels of interface sophistication, while applications like Twitter provide a glimpse of the promise of real-time web applications. Still, implementing real-time web apps has been neither quick nor easy, as conventional web development is hampered by the inelegance and scaling issues of established web frameworks.

The Node project takes a fresh approach to web application development, eliminating traditional barriers to speed and simplicity. Node is an open-source, minimalist server-side JavaScript TCP/IP framework that merges the speed of Google's V8 JavaScript engine (used by the Google Chrome web browser) with two projects, libev and libeio. These projects that provide highly efficient TCP/IP networking capability and "asynchronous" programming support (we'll explain later in this chapter what asynchronous programming is and why it's fundamental to Node's performance). Node, as a result of these design choices, is fast, scalable, and accessible.

A fun example of Node's power is the online game WordSquared (wordsquared.com), a real-time massively multiplayer game similar to Scrabble. WordSquared, shown in figure 1.1, allows many users to play simultaneously on the same huge game board. During gameplay a viewport allows the player to see a small subset of game tiles in detail. Even though WordSquared's virtual game

board has over 50 million tiles, it's common, while playing, to see tiles placed on the board, within the player's viewport, by other nearby players.

WordSquared was originally developed, using Node, over a weekend during the 2010 Node Knockout coding contest. Within the first month of WordSquared's completion 2 million tiles had been played and over 100,000 unique visitors had checked out the game. Although the game leverages a web service called Pusher to provide real-time updates, many Node applications now accomplish the same thing using the Socket.io Node add-on. Both of these update mechanisms leverage the WebSocket protocol which we'll talk about later in this chapter.

Figure 1.1 WordSquared: a real-time multiplayer Scrabble game developed in one weekend, by two developers and a designer, using Node

To hint at why Node is important and innovative, in this chapter we'll talk about Node's place in the history of the Web and how asynchronous programming is conceptually different from conventional programming. We'll also talk about why JavaScript is an ideal choice for Node and why Node is becoming a popular candidate not just for web application development, but for the creation of new TCP/IP protocols, client/server applications, command-line applications, and more. Before we launch into those topics, though, let's turn back the clock to see from whence Node came and why the Web needs Node now.

1.1 Node is moving the Web forward

Node is the most exciting innovation in web frameworks since Ruby on Rails in 2004. Every once in awhile something new happens in web development which changes the way developers think. Rails came into prominance after a screencast showing its use went viral. Rails showed how, by doing things differently than established web frameworks, development could be simplified and developer productivity increased. The ideas in Rails inspired many other frameworks to adopt its methods and web development as a whole evolved. The contemporary ascent of Node is similarly propelled by its radical differences from predecessors. Node's technical and design approach make it a tool capable of advancing the capabilities of the Web.

But it hasn't been easy to get to this point. Looking back at the history of the Web, we see a trend towards increased immediacy in web interactivity and can see that, by helping the Web move in this direction, Node is significant.

1.1.1 The Web before Node

During its early years the World Wide Web resembled, more than anything, a vast, esoteric library. Web pages were static, not interactive. This was to be expected, given that HyperText Markup Language (HTML) was originally conceived as a means of sharing documents.

As time went on, however, the idea of the web-based application emerged. Online discussion forums became widespread. Hotmail, established in 1996, showed Internet users that they didn't need to rely on desktop applications for all their needs. These early web applications were clunky, however: still tied to the web-page-as-document model. Application interaction involved a user filling out a form, submitting it, then having to wait for the web browser to download an entirely new web page reflecting updated content.

In 2004 Google's "Gmail" application was introduced and illustrated the benefits of looking past the web-page-as-document model. Gmail popularized a technique, now familar to web most web developers, called Asynchronous JavaScript and XML (AJAX) that allows the browser to send and receive information without requiring page reloads. Combined with Dynamic HTML (DHTML) techniques, which used client-side browser manipulation to create rich interfaces, AJAX made possible the first generation of quasi-real-time web applications. The launching of Google's AJAX-driven "Maps" application in 2005 further drove home the potential of the Web for interactivity.

In 2006 a now-ubiquitous social networking site called Twitter launched. Twitter allowed users to broadcast short messages to friends using either the web or cell phone (via text messaging). Communication via Twitter was almost instant and as the service grew in popularity, its short, immediate messaging found many uses. Twitter's eventual mainstream success drove home, to many, the importance of the idea of achieving real-time web-based interaction.

As Twitter's growth skyrocketed with mainstream acceptance, however, scaling the service became a challenge. The web community began to think about the technical problem of serving constantly changing content to a large audience.

Figure 1.2 A short history of Web technology. As immediacy of interactivity increases, web pages become more like traditional desktop applications that don't require installation. With Node making real-time interactivity accessible, who knows where the Web will go next?

1.1.2 Real-time Web with Node

In 2009 Ryan Dahl created a framework that appeared to propose an answer to the technical challenge of interacting with a large web audience in real-time. His Node framework (sometimes called Node.js to differentiate it from other uses of the word "node") approached the problem of serving web content in a new way. Node embraced the idea of "event-driven" (also called "asynchronous") programming, a paradigm in which a developer describes how a web application should respond to specific events rather than describing application logic sequentially. We'll explain this concept further later in the chapter.

Node in its entirety depends on community-created modules and core modules, both of which we will talk about later, both of which sit upon the Node core itself, as illustrated in figure 1.3. Node leverages a number of existing open-source projects: most importantly Marc Lehmann's libev and libeio C libraries and Google's V8 JavaScript engine. The libev and libeio C libraries handle the intricacies of event-driven networking and input/output while the V8 engine allows Node to be programmed using JavaScript.

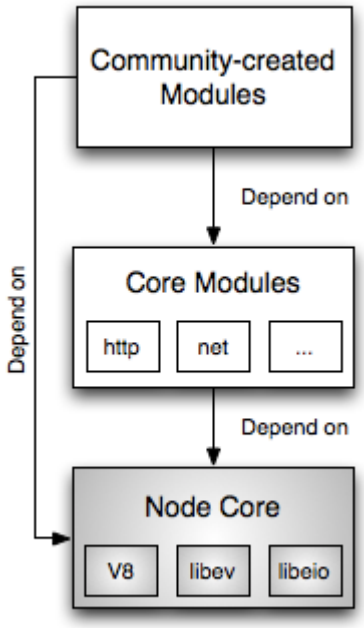

Figure 1.3 Node is conceptually composed of three layers of functionality: the underlying core engine, a number of core modules that add provide utility functions and APIs, and community-created modules for everything else.

V8 was created primarily to provide a fast JavaScript engine for Google's Chrome browser, but V8's developers open sourced it with the intent that it be used by other projects as well. V8's primary innovation is that, unlike traditional JavaScript engines, it compiles, rather than interprets, JavaScript. Compilation in V8 is done on-the-fly, without causing a noticable delay to the user. This technique is referred to as JIT (Just in Time) compilation and is time-tested, having been used by the Java and Smalltalk programming languages. At V8's core is a virtual machine: an emulated abstraction of a computer that can be programmed the same regardless of what hardware it's running on. V8's lead developer also developed, among other things, the Java ME virtual machine which provides Java support for mobile devices.

NOTE **Smalltalk**
Smalltalk is a heavily object-oriented language, originally created for educational use in the 1970s, that has a small but devoted community. Its syntax and design inspired the creators of the better-known Ruby and Python languages. Like Node, Smalltalk runs using a virtual machine.

The combination of asynchronous programming and V8's speed allowed Node to potentially handle a much greater volume of traffic than established web frameworks. The project gained traction in the developer community in late 2009 when well-known Python programmer Simon Willison recognized its potential and became an evangelist. From that point interest grew rapidly and the Node community began to take shape. Node's community is now thriving and well-regarded for its friendly guidance of newcomers and its collaboration and innovation: once a development need has been identified, the community generally self-organizes to take care of it. At the time of writing, add-on modules have been created by over 800 Node enthusiasts. Whatever your development need - whether it be a testing framework, web service API, or database library - chances are the community has probably fulfilled it.

1.2 Node is different

Node is able to move the Web forward because it does things differently than established web frameworks, which were developed using general purpose languages with synchronous programming in mind. Established frameworks are great if you're looking to create an AJAX-driven web application, but they weren't designed for the real-time Web.

Established frameworks are meant to handle traditional Web application needs only; unlike Node, they do not easily accommodate innovations such as WebSocket and they can be challenging to scale. Let's investigate these differences in greater detail to see why they matter on the real-time Web, starting with Node's minimalist design choices.

1.2.1 Node's minimialist design choices

A widely-held value in the Node community, since the beginning, is the love of simplicity and minimalism.

Simplicity is prerequisite for reliability

-- Edsger W. Dijkstra

STARTING FROM SCRATCH

The Node project deliberately started out using a language interpreter that didn't have pre-existing libraries. This was done for a fresh start, so that built-in and community libraries would be built, from scratch, that would be asynchronous. Before Node there existed a number of projects, such as Python Twisted and Ruby Eventmachine, that offered the power of event-driven programming, but were built on top of foundations that weren't designed to be asynchronous. The vast majority of the standard and community libraries for Python and Ruby are made up of synchronous code that can limit the performance gains of any asynchronous logic.

SIMPLE, YET POWERFUL, APIS

Node's decision to start fresh also meant that Node's API design could be informed by the past, delivering something more elegant than the APIs of established frameworks. Node's core modules, illustrated in figure 1.3 earlier, provide a number of simple yet powerful APIs. In addition to Node's `http` module, which provides HTTP client and server functionality, and Node's `net` API, which allows access to raw TCP/IP functionality, there are over 20 other modules that provide a wide range of functionality. As a result, Node can serve as the foundation on which things that go beyond conventional web applications (such as WebSocket, FTP, DNS servers, etc.) can be built.

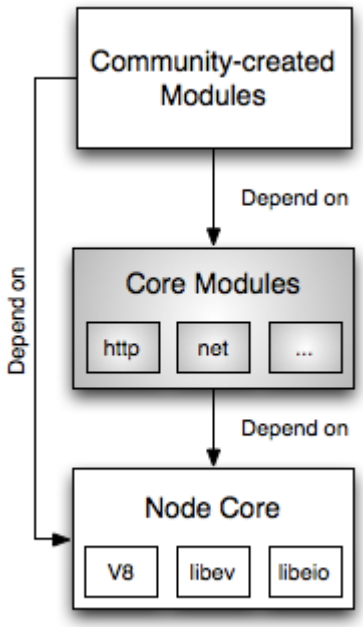

Figure 1.4 Node's core modules provide a wide range of functionality.

SIMPLER HANDLING OF TASK DELEGATION

Node's preference for simplicity, however, extends beyond the API. Most established web application platforms rely on "threads". 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 people who've been doing this a long time, managing threads within an application can be complex. Also, when a large number of threads are needed to handle many concurrent server connections, threading can tax operating system resources. Each thread requires CPU resources to schedule when it can work and each requires a certain amount of RAM.

Node foregoes threads for what is called an "event loop". An event loop lets the framework itself, rather than the operating system, manage switching between tasks. Node uses the libev C library to manage event loop functionality.

MINIMALIST ARCHITECTURE

Node is also minimalist in how it approaches scalability, adhering to a shared-nothing (SN) architecture. This means that each Node instance has its own process and memory. A Node application will normally run on a single CPU or core, but if you want to run Node on multiple CPUs/cores Node's cluster API [1] makes it easy.

Footnote 1 http://nodejs.org/docs/latest/api/cluster.html

1.2.2 The asynchronous advantage

Node's minimalist design choices create a clean foundation for asychronous development, which is the most radical difference you will find when comparing Node to established web frameworks. Understanding asynchronous programming will likely be the biggest challenge you will have in getting a handle on Node development, but once you understand this type of programming it can be engaging and fun.

Asynchronous programming sounds mysterious, but the underlying concept is conceptually simple. For example, in traditional, synchronous programming, a program will initiate a request for a database record and will sit there and wait until the request is fulfilled. In asynchronous programming, however, the program will

initiate a request to the database, specifying what should be done with the request's result. The program will make a note of what is to be done then move on to the next task without waiting for the result to be returned. Only when the database request result returns will the specified result handling logic be triggered.

As an example of this technique in action, imagine a web application which allows the user to manage a to-do list, storing to-dos in a database. When using the web application, a user could submit a form, specifying a task, to the application and the application could, before even completing the insertion of the task into a database, return a web page to the user. Studies have shown[2] that response speed is the most important factor to users and a quick web application response can do a lot to increase a website's "stickiness".

Footnote 2 http://www.websiteoptimization.com/speed/tweak/design-factors/

Asynchronous programs, in short, spend less time waiting around. In the context of programming, unnecessary waiting is referred to as "blocking". Asynchronous logic and applications are referred to as "non-blocking".

Executing an asynchronous program is a bit like cooking a meal. If you were preparing a pasta dish, you wouldn't wait until the water boiled to chop your vegetables and start frying them. You'd simply put water on the stove, turn on the heat, make a mental note of what needs to be done once the water reaches a boil, then immediately move on to chopping the vegetables. By the time the water was boiling, as shown in figure 1.5, you'd likely have the vegetables chopped and could fry them up as the pasta cooks.

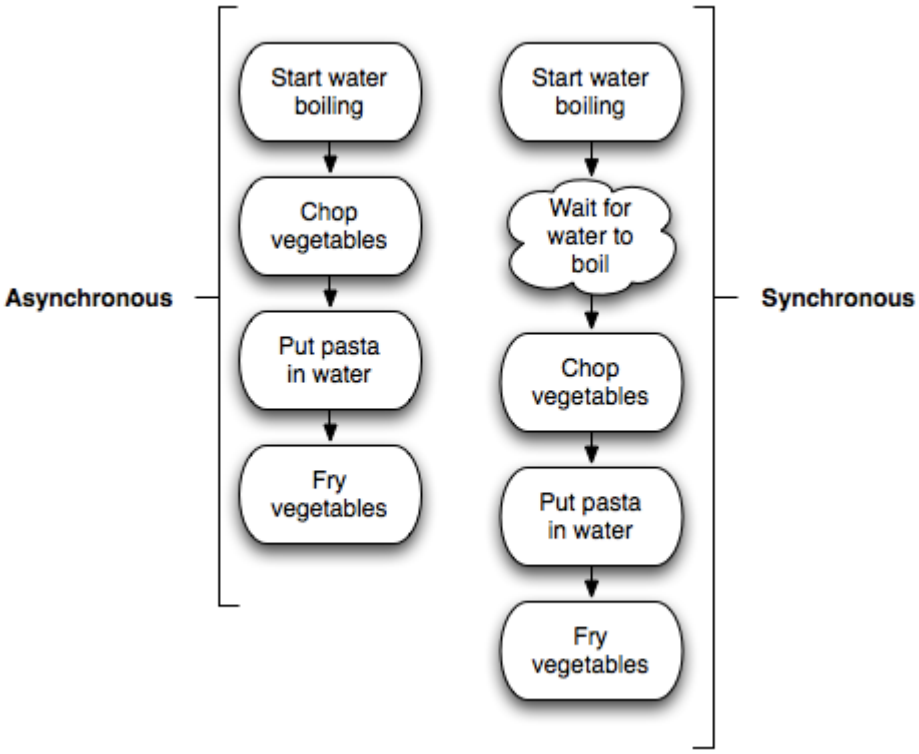

Figure 1.5 Using cooking as an example, asynchronous execution is shown to be more efficient because it requires less time be spent waiting around. Node-based asynchronous development results in efficient applications.

You say you're not a cook? Lets constrast some synchronous JavaScript with its asynchronous counterpart. The code in the following snippet calls a function `getResultsFromDatabaseSync`, defined elsewhere, that attempts to return results from a database. As expected, the first line executes first, the database results are returned next, and only then does the last line execute.

```
console.log('I execute first!');
var result = getResultsFromDatabaseSync(); // executes next
console.log('I execute last!');
```

Asynchronous programming is different. In the next example we call a function `getResultsFromDatabase`, defined elsewhere, that will attempt to get results from a database. `getResultsFromDatabase` is executed given an anonymous function as an argument. The anonymous function would normally determine what we want to do with the results (using the `result` argument), or what to if there was an error getting the results (using the `error` argument), but in our example the response logic simply outputs 'I execute last!' to illustrate the order of

execution. For the purposes of this example, we presume that the process of retrieving data will take a bit of time. Despite the fact we've defined the response logic before outputting "I execute next!", the response logic won't be triggered immediately.

```
console.log('I execute first!');
getResultsFromDatabase(function(error, result) {
  console.log('I execute last!');
})
console.log('I execute next!');
```

Because the sequence of logic is more variable in asynchronous development, programming requires a different mindset. You can employ techniques and third-party add-ons to help manage asynchronous execution at a higher level. In chapter 4 we'll talk about these techniques and add-ons and explain how you can create your own tools for managing execution.

1.2.3 The JavaScript advantage

Now that you've had just a taste of asynchronous development, lets talk about why JavaScript is such a good fit for this style of programming.

Until Node, web frameworks were largely based on languages without built-in support for asynchronous programming: Perl, PHP, Java, Python, and Ruby. While third-party libraries for these languages support event-driven programming, working with these libraries requires additional knowledge, sometimes with considerable learning curves. If you've explored alternatives to Node, you'll likely come to appreciate Node's elegant approach.

> I had a rather sudden epiphany that JavaScript was actually the perfect language for what I wanted: single threaded, without preconceived notions of "server-side" I/O, without existing libraries.
>
> -- Ryan Dahl, creator of the Node project

JavaScript is a language familiar to web development professionals. JavaScript's syntax and naming conventions are influenced by C and Java, but the language is higher-level, meaning it generally takes less lines of code in JavaScript to express the equivalent C or Java logic. Despite JavaScript's comparative brevity, however, it is powerful. In its early years, JavaScript was disparaged due to incompatibilities between browser implementations and difficulty of debugging,

but as time went on implementations improved and the expressiveness of the language gained appreciation.

JavaScript is excellent for specifying asynchronous logic because functions are "first-class objects". This means the language allows functions themselves to be passed as function arguments, used as return values, and assigned to variables.

One common use of functions in Node is as a "callback". Callbacks are used to specify logic to perform upon completion of an asynchronous function. A callback takes the form of an anonymous function or the name of a previously defined function.

Following is an example of a JavaScript function that accepts two callbacks: one describing action to take upon success and the other describing action to take upon failure. This success/failure pattern is fairly common in Node programming. In the example the function is called with two anonymous functions that use Node's `console.log` function to output text. Success or failure is determined randomly.

Listing 1.1 horse_race.js: using multiple callback arguments in a function

```
function horse_race(success, failure) {

  var victory = Math.round(Math.random());

  if (victory) {
    success();
  } else {
    failure();
  }
}

horse_race(
  function() { console.log('Goodness! I won!'); },
  function() { console.log('Drat. I lost.'); }
);
```

Like Ruby and Python, JavaScript is also dynamic. Dynamic languages are high-level and allow "monkey patching": the modification of run-time code during program execution.

NOTE **Monkey patching**

Monkey patching is a powerful programming technique for dynamic languages that you can take advantage of when developing in Node. In non-dynamic, static languages, once a function or class has been defined, it can't be altered during program execution. In dynamic languages, you can simply redefine a function or class. If you've previously developed using languages that don't easily allow this, like PHP or Java, the capability may seem unusual, but in practice you'll likely grow to love the flexibility it provides.

JavaScript is also a perfect companion to Node because it has the advantage of being portable between the browser and the server. As long as any APIs accessed by logic are available on both browser and server, the same logic should run in either environment. Form validation, for example, is often duplicated in the client and server. Why write the same code twice? Reusing logic in both the browser and server enforces consistancy and means not having to reinvent the wheel. It also means you have to maintain less code. Another benefit you'll find when working in Node is you don't have to mentally context switch between languages when working on different parts of a web application. Staying in one language is likely to increase your productivity and make development more enjoyable.

1.2.4 Advancing the idea of the application server

In addition to the concept of asynchronous development and server-side JavaScript, another idea that may be new to you if you come from a PHP background is the idea of writing a web application server instead of page scripts. Unlike PHP, developing in some web development frameworks, such as Ruby on Rails or the Python-based Django web framework, involves writing a server that sends and receives HTTP. Web development using Node follows the same principle.

Application servers give you more control than writing PHP scripts that get triggered by a web server such as Apache. Writing a web application with clean URL support in PHP requires you to use web server rewrite functionality (such as Apache's mod_rewrite). Rewrite functionality generally translates the path portion of the URL into a global variable PHP can access. Most web application servers require no such hacks.

As opposed to writing a PHP script, when writing a Node web application server we interface primarily with a request and a response object. The request

object contains details about the web client making the request and what they are requesting. The response object contains methods that allow you to send data to the web client. Many frameworks exist that leverage the request and response objects to provide functionality commonly needed in web applications. The most popular, called Express, we cover in chapter 8.

Node add-on frameworks generally include functionality to associate URL patterns with related logic, provide session support, help render data to HTML, and allow media files, such as images, to be served as content. These frameworks generally follow the time-tested Model-view-controller (MVC) design pattern.

NOTE

MVC

The MVC design pattern is a way of organizing applications that separates data, data manipulation logic, and output. In the context of web applications, the "model" is usually manipulated using a database API or an object that provides a higher level interface to data. The "view" portion of the architecture is usually represented by templates. The "controller" is the logic that manages the flow of data from the model to the view (and input from the view back to the model). If you don't have experience with MVC, Jeff Atwood's online article "Understanding Model-View-Controller"[3] provides a good introduction to the concept. If you don't get it immediately, don't worry: you'll likely gain an understanding of the concept as you read through this book.

Footnote 3

http://www.codinghorror.com/blog/2008/05/understanding-model-view-controller.html

Now that you've seen how Node is different from established frameworks, in its design choices and use of asynchrous programming and JavaScript, let's look further at one more reason why the web of today and tomorrow needs Node: Node's ability to do more.

1.3 Node can do more

Node's capabilties go beyond that of traditional web frameworks. A Node web application, for example, can easily integrate support for non-HTTP protocols, such as WebSocket. Node is flexible and powerful enough that it can be used to implement TCP/IP applications such as a reverse HTTP proxy, caching web requests from a slower, non-Node web server (emulating the functionality of special-purpose applications like Squid and Varnish). Node can even be used to develop entirely new TCP/IP protocols.

In this section we'll look at a few ways Node is being used to develop non-web applications. While you may be predominantly interested in Node's use for web application development, knowing the full spectrum of Node's capabilties may lead you to use Node in unexpected ways.

1.3.1 Server applications

Node has been used to implement versions of a variety of traditional server applications. Node-based DNS servers, web crawlers, message queue servers, and many other types of applications exist.

Server applications - such as file transfer, email delivery, and instant messaging applications - have traditionally been created in non-dynamic languages such as C, C++, and Java. C is a very flexible and powerful language, but is comparatively low-level and can be challenging to learn. C++ significantly extends C, but offers a cornucopia of language features that, if not used correctly, allows you to "shoot yourself in the foot" (the ability to redefine operators like "+" is one example). Java has fewer sharp edges than C++, but is more verbose than C and less flexible than dynamic languages.

As we mentioned earlier, Node breaks from tradition and leverages JavaScript which makes it accessible to a large number of programmers. Given JavaScript's strengths, if Node is widely adopted by server developers the result will likely be quicker application development and increased diversity in server software.

Browserling, shown in figure 1.6, is an example of a Node-based website that makes great use of Node's non-web capabilities behind the scenes. The site allows in-browser use of other browsers, including the notorious Internet Explorer 6. 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. Browserling has VMs run test browsers and StackVM then relays video and keyboard/mouse input data between the user's browser and the emulated machines.

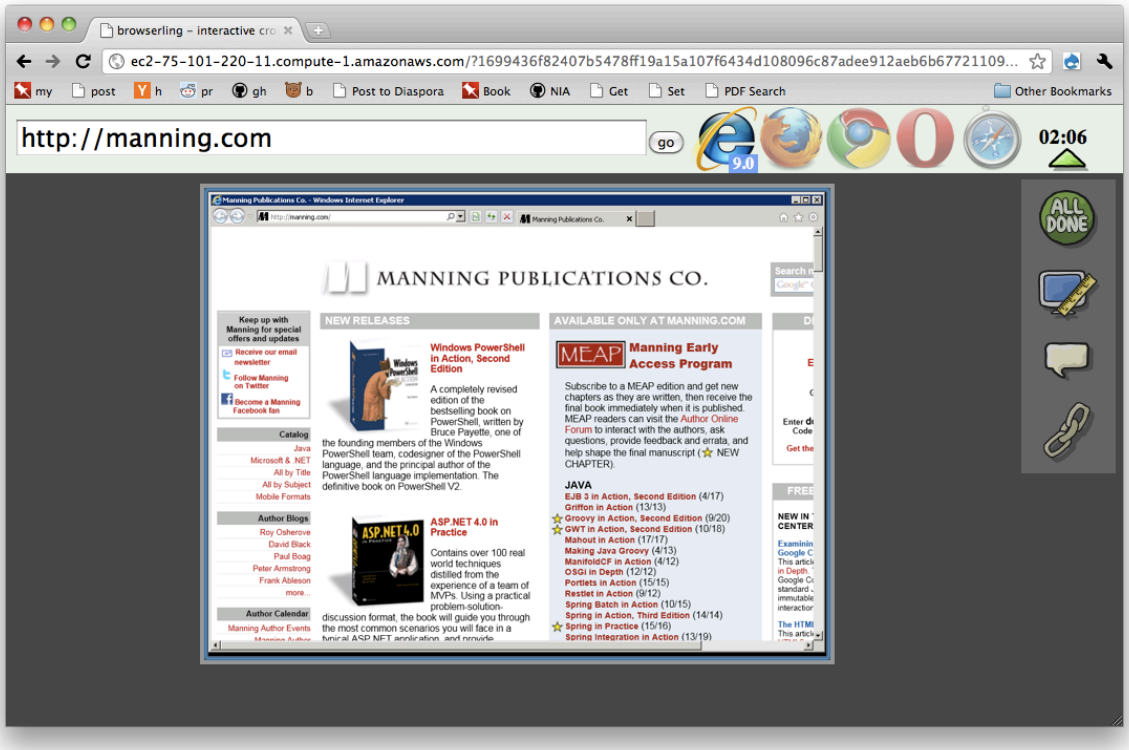

Figure 1.6 Browserling is a web application, predominantly used for cross-browser testing, that uses the Node-driven StackVM server application to manage virtual machines running browsers.

1.3.2 Protocol development

Servers leverage underlying TCP/IP protocols and Node is a great tool for developing new protocols.

While the Web's native protocol, HTTP, is well suited to transporting documents, it is not the perfect fit for all the Web's needs. A recent example of improving the Web with new protocols is the WebSocket protocol. The Web's first real-time applications used AJAX to exchange messages with the web server. AJAX uses HTTP as a transport mechanism to send messages back and forth to the server. Because HTTP requires a lot of communication overhead, AJAX's performance is limited and both server and client have to do extra work. WebSocket was created in 2010 to provide an alternative: a lightweight, bidirectional protocol specifically focused on real-time textual communication. The WebSocket standard is still under development, but a number of browsers

support it. The popular Socket.io project, which implements a robust Node-driven WebSocket server, is a good example of Node being used with emerging protocols.

If you're a die-hard dreaming about trying out protocol development, Node may be just the technology you've been waiting for.

1.3.3 Command-line applications

The things that make Node perfect for server and protocol development also make Node a great candidate for the development of command-line interface (CLI) applications. Examples of common CLI applications include version control applications, database clients, and compilers. In addition to its networking API, Node offers access to functionality commonly needed by CLI applications: filesystem, process, and operating system functionality.

The Node community has created modules that make CLI appliction development easy by elegantly handling option and command parsing. The combination of the familiar JavaScript language with Node's elegant APIs and easy CLI parsing has the potential to unleash the creativity of many who have, until now, only dabbled in CLI application creation.

1.3.4 Screen scraping

A less traditional type of application that Node is perfect for is the "screen scraper". Screen scraping is the art of parsing web pages to extract data. Node is, more than another other web framework, suited for this task because Node is capable of emulating web browser functionality.

Because Node is built using Google's V8 JavaScript engine, Node speaks the same logical language as the browser. JSDOM, a Node community add-on, leverages this capability, allowing Node to emulate the web browser's Document Object Model (DOM).

The DOM is essential for web browser emulation, acting as a JavaScript-accessible interface to HTML/CSS content. By emulating the DOM, Node can read and write HTML in a virtual browser. Node can then take advantage of high level JavaScript libraries, like JQuery, that allow high-level manipulation and querying of the DOM.

Following is an example of using JQuery and a community-created Node module to traverse the DOM in a virtual browser:

```
var jsdom = require("jsdom");

jsdom.env("http://releases.ubuntu.com", [
```

```
  'http://code.jquery.com/jquery-1.5.min.js'
], function(errors, window) {
  window.$('ul:first > li').each(function (index) {
    console.log(window.$(this).text())
  })
});
```

DOM emulation is not only useful for screen scraping, but for automated web application testing as well. The "Zombie.js" project uses Node to implement an automated testing framework that allows client-side JavaScript to be tested without needing a browser.

And finally, for ad-hoc testing/monitoring, the Node community has also created a tool, called "query", that provides command-line access to JQuery functionality.

1.4 Summary

If you're a web developer willing to take the time to explore Node, we think you're going to end up thinking of web development in a new way - and you'll greatly enjoy yourself in the process. Not only will you arm yourself with an elegant, scalable tool that allows you to create real-time websites and develop non-HTTP applications, but you'll also prepare yourself for where the web will be tomorrow.

In upcoming chapters, we'll give you a solid foundation for Node development, guiding you through the development of web applications and beyond. We'll teach you how to deal with asynchronous programming challenges, how to leverage high level frameworks, and how to go beyond web application development, touching on how to develop TCP/IP servers and command-line applications.

Before we get to all that, though, we first need to look at what you should know before developing in Node, how to install Node and community add-on modules, and how to create your own modules. We'll look at these topics in chapter 2.

2 Getting started with Node

In this chapter:

- Installing Node
- Creating your first Node programs
- Installing community add-ons
- Organizing and reusing Node functionality

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.

By then end of this chapter you'll have installed Node and started experimenting. In this chapter we'll run through examples of a few types of Node development and will explain how to install community add-ons using the Node Package Manager tool. Finally, we'll explain the use of Node "modules", Node's way of keeping code organized and packaged for easy reuse.

Before installing Node, however, we'll look at what you'll need to know to get the most out of Node development.

2.1 Preparing for Node development

When learning to develop web applications in Node, experience working with established web development frameworks is useful. You'll benefit from a working knowledge of Hypertext Transfer Protocol (HTTP), HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and the Model-view-controller (MVC) design pattern. We're not going to cover these topics here, but we will explain the role of two other important technologies: JavaScript and the command-line.

If you've got a good handle on JavaScript and the command-line, you're ready to move on to Node installation. If not, we'll run through what you need to know and why.

2.1.1 Sharpening the JavaScript saw

When JavaScript first rose into prominence it didn't receive an entirely warm reception. Incompatibility between browsers and a lack of high-level libraries meant that JavaScript development involved frequently bumping into sharp edges.

Part of JavaScript's dubious reputation also had to do with early uses of the language. JavaScript was initially used for mundane things: client-side form validation, image "rollovers", and assorted interface parlour tricks. It wasn't until the emergence of Dynamic HTML and AJAX that JavaScript's role began to be taken seriously.

If your knowledge of JavaScript mostly consists of casual client-side manipulation, you may want to take some time to re-aquaint yourself. When developing with Node you'll be using JavaScript on the server-side as well as the client-side so it's worth taking some time to "sharpen the saw".

Fortunately, there are a number of online resources and good books for contemporary JavaScript enthusiasts. The Mozilla Foundation's "A re-introduction to JavaScript"[1] was written by Node enthusiast Simon Willison and provides a comprehensive online language refresher. Douglas Crockford's "JavaScript: The Good Parts" is a popular, relatively short book that focuses on contemporary use of the language. Manning Publications' "Secrets of the JavaScript Ninja", written by JQuery creator John Resig, contains practical details on advanced applications of JavaScript.

Footnote 1 https://developer.mozilla.org/en/A_re-introduction_to_JavaScript

2.1.2 Getting familar with the command-line interface

In addition to refreshing your JavaScript knowledge, it's also worth making sure you've got a good grasp of using your operating system's command-line interface (CLI). For those unfamiliar, a CLI allows you to navigate and manipulate the filesystem by entering commands on the keyboard instead of relying on a mouse-driven graphic user interface (GUI). Windows comes with its own flavor of command-line interface and Mac OS X, Linux, and Cygwin (a Unix-like environment that is run in Microsoft Windows) come with a more traditional "Unix" flavor that supports the Portable Operating System Interface for Unix (POSIX).

Even if you're using Windows, you might want to read up a little on Unix commands. Knowing basic Unix commands is useful because Node's APIs support commonly used POSIX functionality, like deleting and renaming files, and allow you to run external command-line utilities for anything Node itself doesn't handle. Even if you're only writing web applications, you'll likely need this functionality as many web applications involve some interaction with the server's filesystem: whether storing files uploaded by users, resizing images on-the-fly, or writing to log files. The online tutorial "Learn Unix in 10 Minutes"[2] is a great resource if you're just starting out learning Unix commands.

Footnote 2 http://freeengineer.org/learnUNIXin10minutes.html

If you're not familar with command-line interfaces, it may be worth taking some time to familiarize yourself. If you run OS X, the Terminal utility provides a CLI from which to explore Apple's variant of Unix. If you use Ubuntu, you can access a CLI by using the Terminal utility in the Applications/Accessories menu. If you use Microsoft Windows, the standard "Command Prompt" can usually be found in the Accessories section of your application selector.

During Node development, the command-line will likely become familar territory. The Node Package Manager (npm) is a handy command-line tool for extending Node with add-on modules. Many Node modules also come with their own command-line tools for ad-hoc access to module functionality. Node itself, however, can be installed on the most popular operating systems without using the command-line.

2.2 Installing Node

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 Node installation on the OS X, Windows, and Linux operating systems.

2.2.1 OS X setup

Installing Node on OS X is quite straightforward. The official installer[3], shown in figure 2.1, provides an easy way to install a precompiled version of Node and npm.

Footnote 3 http://nodejs.org/#download

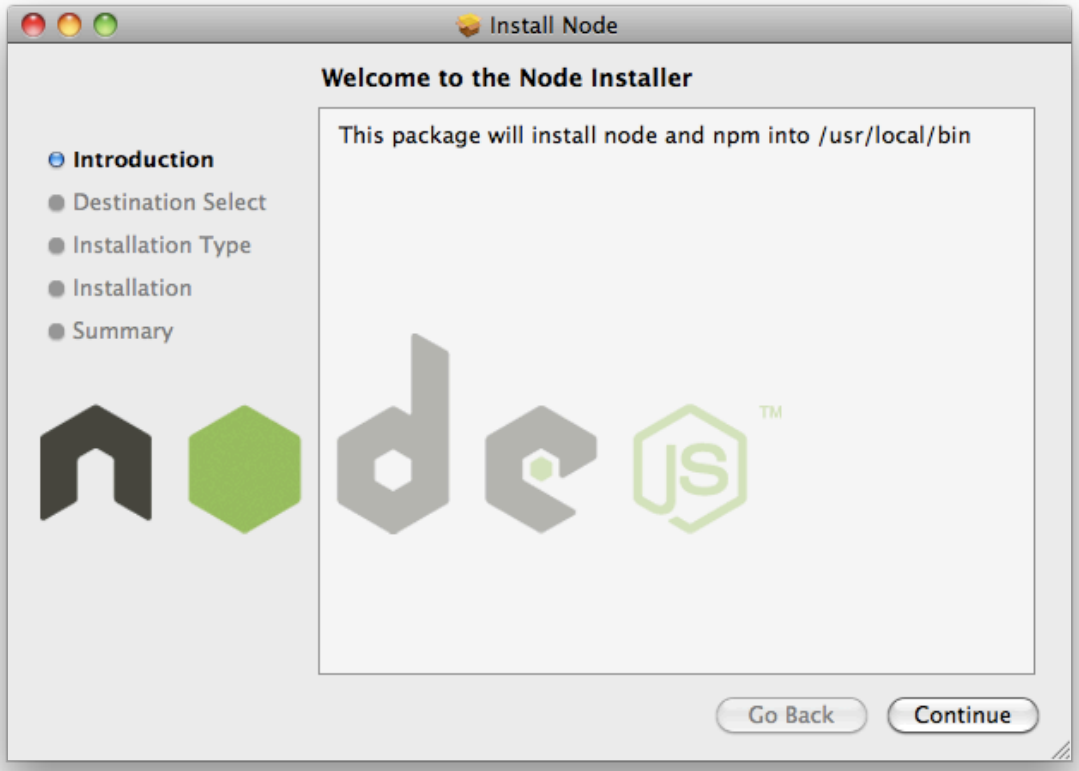

Figure 2.1 The official Node installer for OS X

If you'd rather install from source, however, you can either use a tool called Homebrew, 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.

NOTE **XCode**

If you don't have XCode installed, you can download XCode from Apple's website[4] (note that XCode is a large download: approximately 4GB).

Footnote 4 http://developer.apple.com/xcode/

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 2.2.4 "Compiling Node", later in this section, for the necessary steps.

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 -fsSL
➥https://gist.github.com/raw/323731/install_homebrew.rb)"
```

Once Homebrew is installed you can install Node by entering the following:

```
brew install node
```

As Hombrew compiles code you'll see a lot of text scroll by. The text is information related to the compiling process and can be ignored.

2.2.2 Windows setup

Node can be most easily installed on Windows by using the official stand-alone installer[5]. After installing you'll be able to run Node and npm from the Windows command-line.

Footnote 5 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-like environment. You'll likely want to avoid this way of installing 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. If you'd like to install Node using Cygwin, please see "Appendix A: Setting up Cygwin in Windows for Node Development".

2.2.3 Linux setup

Installing Node in Linux is usually painless. We'll run through installation, from source code, on two popular Linux distributions: Ubuntu and Centos.

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 g++ libssl-dev git
```

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.

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.

2.2.4 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 then 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 (substitute the "node-v0.6.1" with whatever the directory name is for the current version of Node):

```
cd node-v0.6.1
```

You are now in the directory containing Node's source code and, from here, can easily compile it. To do so, enter the following:

```
./configure
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
```

You should now have Node on your machine! Enter the following to run Node and have it display its version number:

```
node -v
```

2.3 Creating your first Node programs

Now that you've installed Node on your machine, you can begin to have fun with it. The traditional "hello world" example is always satisfying, indicating the point in approaching a new technology when you finally get to interact with it. As Node is used in a number of ways - interactively, as a script, or to create server and web applications - we'll run through "hello world" examples in the four development contexts.

We chose these four contexts because they are the most common, and useful, contexts. Running Node interactively is helpful for learning and experimenting. Running Node as a script is the foundation of building any kind of application: be it an automated process, command-line tool, or server. Creating a TCP/IP server is the foundation for protocol development. Creating a web server is the foundation for the most common use of Node: web applications.

As interactive programming is the quickest way to try out Node, we'll start by diving into Node's read-eval-print loop (REPL): an interactive command-line tool useful for experimenting with the framework.

2.3.1 Node's interactive mode

The Node REPL allows you to enter Node logic, line by line, then see the returned value of each line entered.

To start the REPL, open up a command-line on your system then enter `node`. You should then see a ">" prompt.

At the prompt, type the following:

```
console.log('hello world')
```

Node then executes your code, displaying the following:

```
hello world
```

To exit from the REPL, enter CTRL-d at any time or type `process.exit()`. While using the REPL, the variable "_" will automatically be assigned the value of the last expression evaluated. This is handy as it reduces typing.

```
> 3 + 4
7
> _ * 2
14
```

Whenever you're stuck on how Node functionality works, trying it out using the REPL is a great way to clear things up.

TIP	**Keyboard Shortcuts** The UP and DOWN keys can also be used to cycle through commands previously entered and CTRL-A and CTRL-E will move you to the begining and end of your input, respectively.

2.3.2 Node script example

While the REPL is great for experimentation, real-world Node development generally takes place in text files, often referred to as "scripts".

For your first script, create a text file in the directory in which your command-line starts. Add the following Node logic to the script and save it using the filename "hello.js":

```
console.log('hello world');
```

To execute this script, open up a command-line and enter the following:

```

```

```
node hello.js
```

Node then executes your code, displaying the text "hello world".

2.3.3 Hello net example

Now that you've got Node working interactively or as a script, let's try using some networking API calls. Node makes it easy to experiment with basic TCP/IP. In five lines you can create a basic TCP/IP server that does nothing but say "hello net world" to the connecting client.

To try this out, create a text file in the directory in which your command-line starts. Enter the following Node logic into your script and save it, using the filename "net.js", into the directory in which your CLI normally starts:

```
var net = require('net');
var server = net.createServer(function (client) {
  client.end('hello net world\r\n');
});
server.listen(8000, '127.0.0.1');
```

To execute this script, open up a command-line and enter the following:

```
node net.js
```

Your server is now running. To connect to it, enter the following:

```
curl telnet://127.0.0.1:8000
```

As shown in figure 2.2, you should see the text "hello net world" returned.

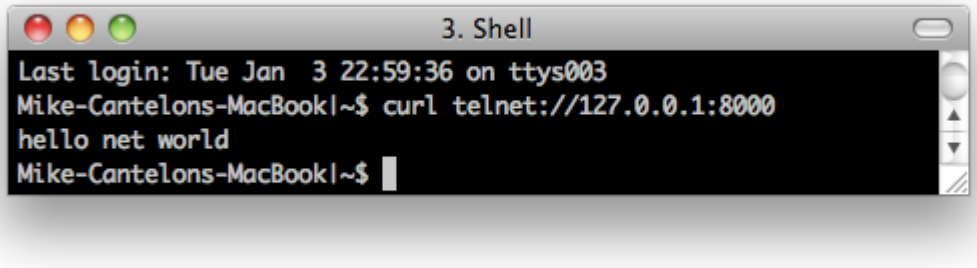

Figure 2.2 Results of connecting to our "hello net" server.

A TCP/IP server in only five lines? Not bad! By creating your own TCP/IP servers you can implement existing protocols or design your own. Writing a TCP/IP server has traditionally been non-trivial as servers have been written in C, C++, and Java: less accessible languages than JavaScript. Node now gives any JavaScript developer the ability to try out a traditionally inaccessible realm of programming.

2.3.4 Hello browser example

Our final example showcases the simplicity of Node's HTTP API. In five lines of code you'll create a very limited web server.

Create a text file in the directory in which your command-line starts. Enter the following Node logic into your script and save it using the filename "http.js":

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('hello world\n');
}).listen(8000, "127.0.0.1");
```

To execute this script, open up a command-line and enter the following:

```
node http.js
```

Open up a web browser and navigate to http://127.0.0.1:8000/. The resulting web page will show the text "hello world". These five lines of code are the fundamental logic used to create web applications using Node. In later chapters we'll build upon this and explore the full potential of Node web application development.

2.4 Installing community add-ons

You've now had a chance to look at what you can do with Node's built in APIs. These are referred to collectively as the Node "core" and include globally available functionality (`console.log` for example) and a number of modules that can be used optionally. Node's core encompasses a lot of useful functionality, but you'll likely want to use community-created functionality as well. Figure 2.3 shows, conceptually, the relationship between the Node core and add-on modules.

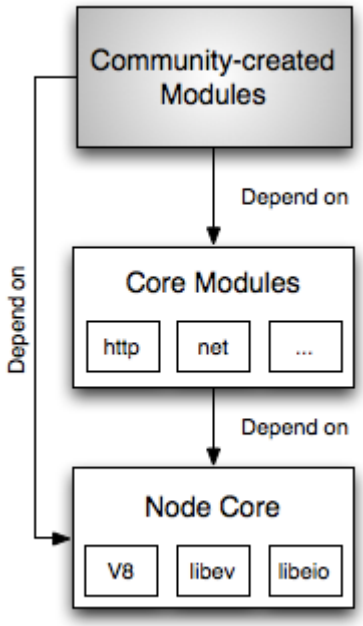

Figure 2.3 The Node stack is composed of globally available functionality, core modules, and community-created modules.

Depending on what language you've been working on, 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 install npm and how to use it to find community add-ons, view add-on documentation, and explore the source code of add-ons.

2.4.1 Installing the Node package manager

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.

Npm 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.

If you've installed Node on OS X or Windows using the stand-alone installer, then npm is already installed and you can skip to section 2.4.2.

On other operating systems, or if using Cygwin in Windows, open a command-line and enter the following to install npm.

```
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 Github[6] where the latest installation instructions can be found.

Footnote 6 http://github.com/isaacs/npm

2.4.2 Searching for packages

Once npm is installed, 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 interface[7] to the repository. This website, shown in figure 2.4, 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 7 http://search.npmjs.org/

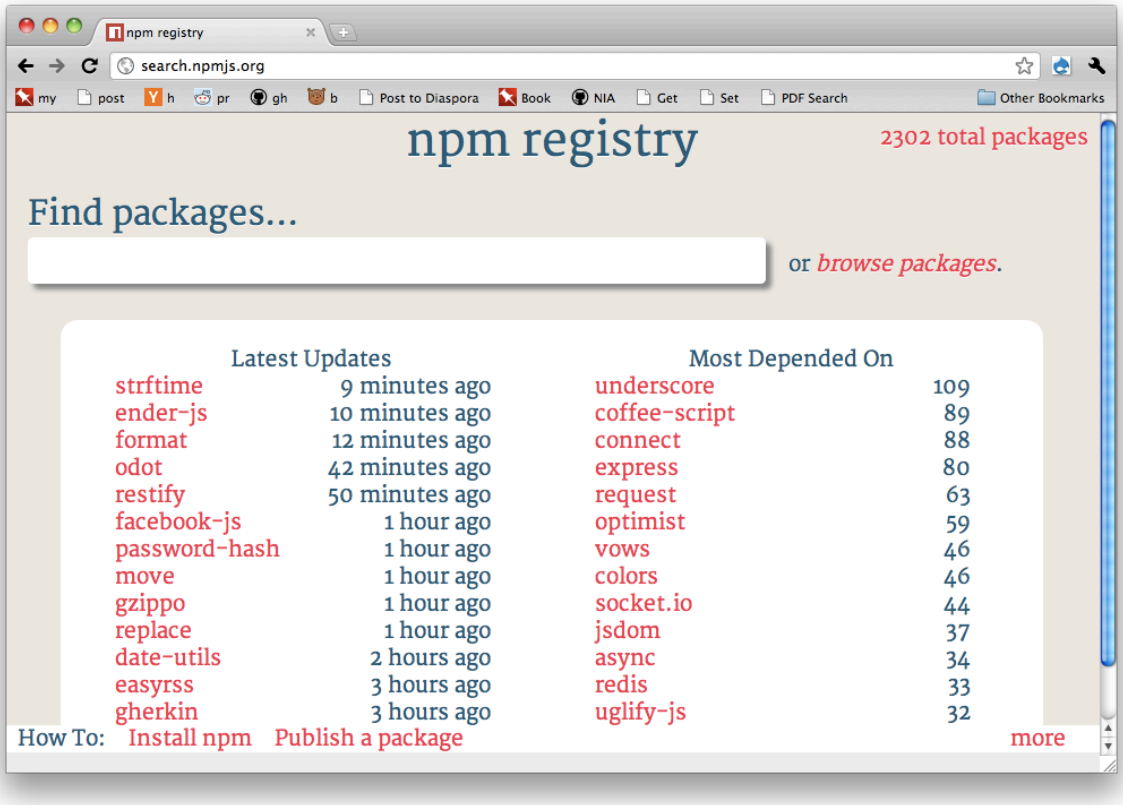

Figure 2.4 The npm search website provides useful statistics on module popularity.

Npm's 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.

2.4.3 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 when 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.

2.4.4 Exploring Documentation and Package Code

Npm 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. Npm provides an easy way to spawn a new command-line 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.

2.5 Organizing and reusing Node functionality

Now that you've installed Node and npm and have worked through some simple examples we'll talk about a problem you'll inevitably face during application development. 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 2.5, is to group related logic into separate files.

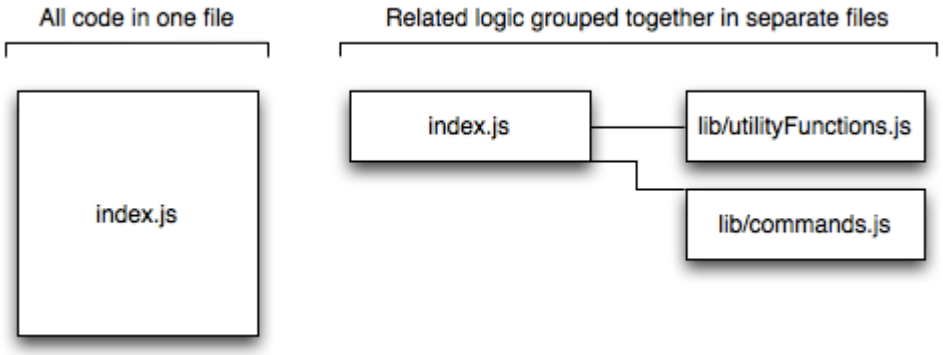

Figure 2.5 It's easier to navigate your code if you organize it using directories and separate files rather than keeping your application in one long file.

In many language implementations incorporating the logic from another file (we'll call this the "included" file) means 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(strtolower($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()
```

Node's module system provides an elegant solution to this problem. 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 returned to the application. These functions and variables are returned as properties of a single object called "exports" or, for when something other than an object needs to be returned, "module.exports". Figure 2.6 shows how this works visually.

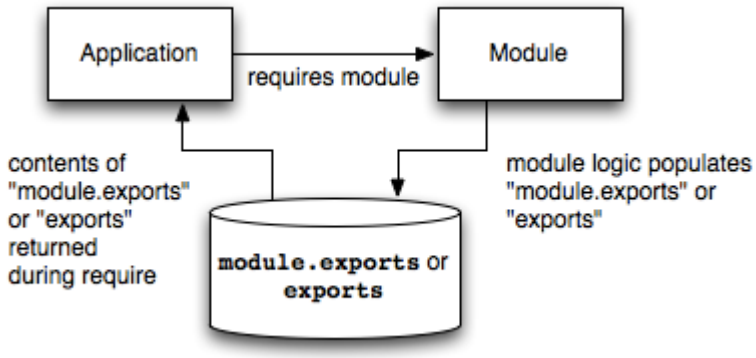

Figure 2.6 The population of "module.exports" or the "exports" object allows a module to select what should be shared with the application.

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 repository 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 12.

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.

2.5.1 Creating modules

Modules can either be single files or directories containing one or more files. 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 2.5.3 "Caveats").

Figure 2.7 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:

```
var canadianDollar = 0.91;

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

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

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

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. The path to the module that you give to the `require` function begins with "./", specifying that the module exists in the same directory as our application script. This means that if you were to create your application script named "test-currency.js" in the directory "projects/currency_app", then your "currency.js" module file, as represented visually in figure 2.8, would also need to exist in the "projects/currency_app" directory.

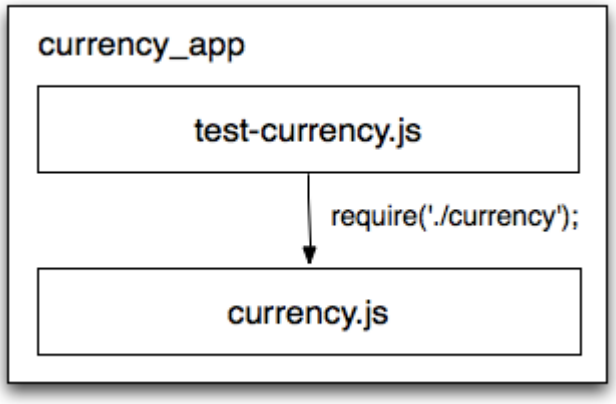

Figure 2.8 When putting a "./" at the beginning of a module require, Node will look in the same directory as the program file being executed.

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.

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

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

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

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.

2.5.2 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. 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));
```

To create a module that would work with the this code you'd want the module to export a JavaScript class implementation rather than an object containing multiple functions. Unfortunately, simply setting the `exports` object to a Javascript class implementation won't work. The following module code attempts to do just this and, when you try to use it with the previous code, a `TypeError: object is not a function` exception will be raised.

Listing 2.1 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`. `module.exports`, in fact, overrides `exports`. 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.

2.5.3 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 table 2.3, with which it will search for this module.

Table 2.1 Steps to finding a module

Step		Yes	No
1	Is the module a core module?	Return module	Continue to next step
2	Is there a "node_modules" directory in the current directory (starting in the same directory as the program file)?	Return module	Continue to next step
3	Move to parent directory if one exists. Does a parent directory exist?	Go to step 2	Continue to next step
4	Does the module exist in one of the directories specified by the NODE_MODULES environment variable?	Return module	Throw exception

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.

2.5.4 Caveats

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

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 2.9 shows a flow chart summarizing these rules.

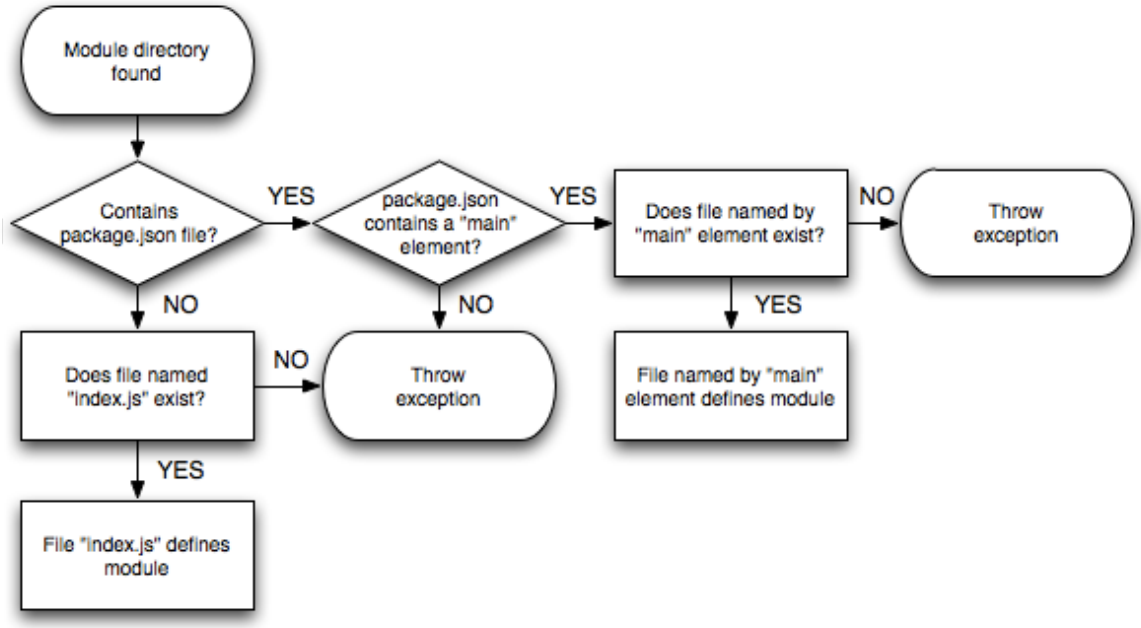

Figure 2.9 The "package.json" file, when placed in a module directory, allows you to define your module using a file other than "index.js".

Below is an example of an "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.

2.6 Summary

In this chapter, we've introduced the fundamental knowledge needed to dive into Node development. We've explained the importance of familiarity with JavaScript and the command-line interface and explained how to install Node on popular operating systems.

The "Hello World" examples in this chapter have given you a taste of the most common Node development contexts. In chapters 4, 5, and 6 we'll teach you how to expand on the "Hello Web" example to create well-organized web applications that can handle the creation, updating, and deletion of data. In chapter 9 we'll expand on the "Node Script Example", teaching you how to create command-line utilities, and the "Hello Telnet" example, teaching you the nuts-and-bolts of TCP/IP server creation.

By explaining how to install and use the Node Package Manager, you're now able to go beyond Node's built-in functionality and leverage the wealth of community contributed add-ons. In chapter 8, we'll teach you how to use a number of add-ons to speed up the process of web application development and do real-time interaction with web browsers using WebSocket.

Now that we've started interacting with Node we'll dive into the most engaging and challenging aspect of Node development: asynchronous programming and testing.

3 Asynchronous programming

In this chapter:

- Handling one-off events with callbacks
- Handling repeating events with event emitters
- Implementing serial and parallel control flow
- Leveraging flow control tools

Asynchronous programming requires a different kind of thinking. 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 machine to those new to it. It's worth taking the time, before beginning development, to learn how you can elegantly control your application's behavior.

In this chapter you'll learn a number of important asynchronous programming techniques that will allow you to keep a tight reign 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. First, however, we'll talk about the pitfalls that developers tend to first encounter during asynchronous development.

3.1 Asynchronous development challenges

When creating asynchronous applications you have to pay more attention to how your application flows and keep a close 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 asychronous logic that hasn't completed processing. As long as there is asynchronous logic that hasn't completed, the Node process won't exit. A continually running Node process is desirable behavior for something like a web server, but isn't desirable for applications like command-line tools. The event loop, for example, will keep track of any database connections until they're closed, preventing Node from exiting.

Application variables can also change unexpectly if you're not careful. The following example shows 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 3.1 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 little bit of JavaScript trickery. In the following code, 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, `color` 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 3.2 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've got a handle on how to keep the event loop free and control your application state, we're going to look at how to cleanly structure your asynchronous application logic.

3.2 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 which trigger response logic. There are two popular models in the Node world for managing response logic: callbacks and event emitters.

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 just store the results in memory for later reference.

Event emitters, on the other hand, provide a framework for organizing callbacks. Event emitters define behavior by specifying response logic that executed when a given event type occurs: either once or every time the event occurs. The different event types an event emitter supports are often conceptually

related, which makes event emitters useful for organizing and reusing code. Node's HTTP server, for example, is implemented as an event emitter as it has to use the same callback logic to repeatedly respond to requests.

So now that we've established that response logic is generally organized in one of two ways in Node, let's jump right into it by learning first how to handle one-off events with callbacks, next how to respond to repeating events with event emitters.

3.2.1 Handling one-off events with callbacks

A callback is an anonymous 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, but because callbacks are comparatively simple a short discussion of them will suffice.

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 database, assembles an HTML page containing the titles, then sends the HTML page to the user. It will create results similar to figure 3.1.

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

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

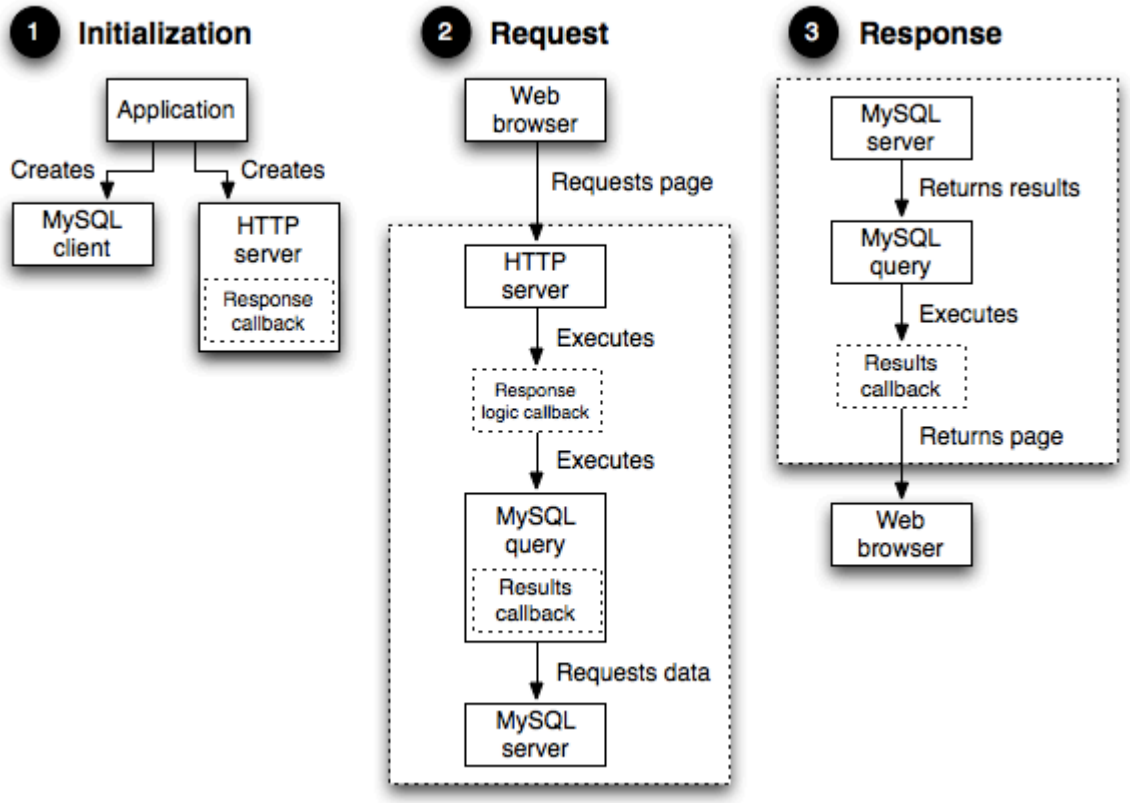

Figure 3.2 How callbacks are used to define response logic in a simple application.

Listing 3.3 contains the code for this example application.

Listing 3.3 blog_recent.js Example showing the use of callbacks

```
var http = require('http')
  , mysql = require('mysql');                                    1

var client = mysql.createClient({                                2
  user:     'myuser',
  password: 'mypassword',
});

client.useDatabase('blog');                                      3

http.createServer(function (req, res) {                         4
  if (req.url == '/') {
    client.query(                                                5
      "SELECT * FROM posts \
        ORDER BY timestamp DESC \
        LIMIT 5",
      function(err, results, fields) {                           6
        if (err) throw err;

        var output = '<html><head></head><body>' +               7
```

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

        res.writeHead(200, {'Content-Type': 'text/html'});
        res.end(output);                                          ❽
      }
    );
  }
}).listen(8000, "127.0.0.1");
```

❶ Require third-party MySQL client module
❷ Connect to MySQL
❸ Use the "blog" MySQL database
❹ Create an HTTP server and use a callback to define response logic
❺ Query the MySQL database for the titles of the 5 most recent blog posts
❻ Use a callback to define what to do when MySQL results are returned
❼ Assemble an HTML page showing the blog titles
❽ Send the HTML page to the user

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

Listing 3.4 multiple_callbacks.js: an example of the use of multiple callbacks as arguments to a single asynchronous function

```
function doSomething() {
  return true;
}

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

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

```
);
```

Using anonymous functions as callback arguments can be messy. The following example has three levels of nested callbacks. Three levels isn't bad, but once you reach, say, seven levels of callbacks then things can look quite cluttered.

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

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

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

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

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

someAsyncFunction('data', outerLogic);
```

Now that you've learned how to use callbacks to handle one-off events - used to define responses for database queries, web server requests, reading files and more - we're going to move on to how to handle repeated events using event emitters.

3.2.2 Handling repeating events with event emitters

Event emitters are entities suited to responding to repeating events with asynchronous logic. Support for them is built into Node and some important Node API components - such as HTTP servers, TCP/IP servers, and streams - are implemented as event emitters.

Event responses are defined 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.

If, for example, you wanted 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.

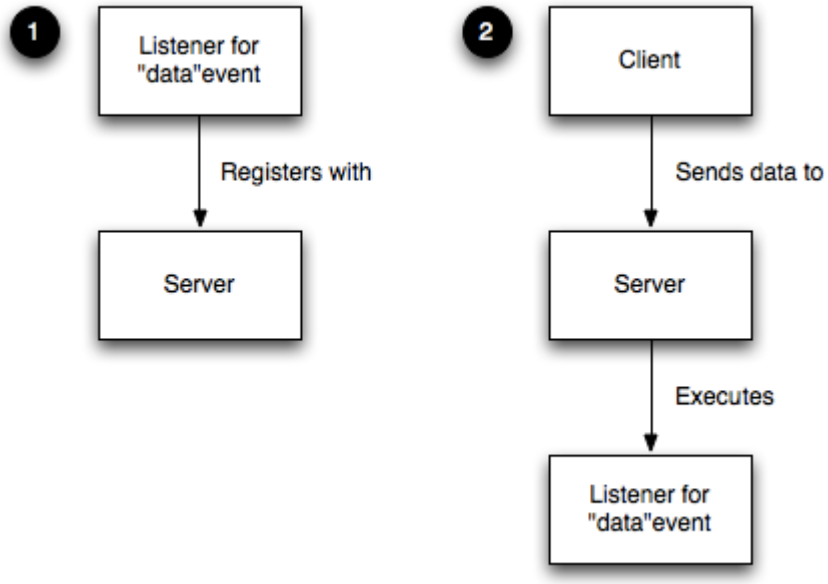

Figure 3.3 How an event listener can be used to act on data received by a Node server.

You could use this technqiue to create an echo server that, when you send data to it, will echo the data, as shown in figure 3.4.

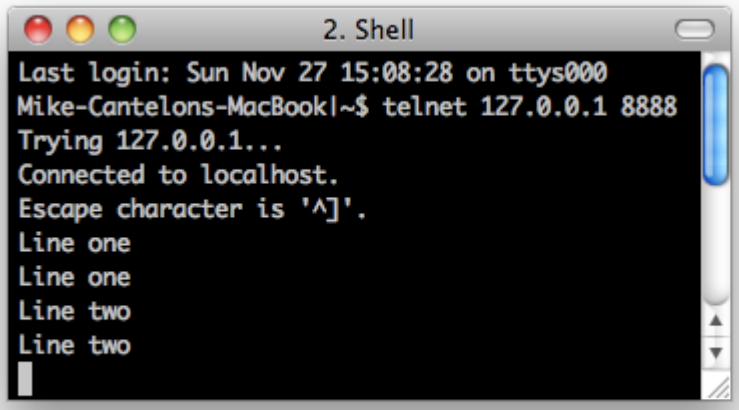

Figure 3.4 An echo server repeating data sent to it.

Listing 3.6 includes the code needed to implement an echo server as an event emitter with a listener defined to respond to "data" event types.

Listing 3.6 echo_server.js: A simple 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);
```

Run this server by entering the following command.

```
node echo_server.js
```

Listeners can be defined that repeatedly respond to events, as the previous example showed, or listeners can be defined that respond only once. The following code 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);
```

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 a 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!");
});
```

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

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

Going farther with this, you can create simple publish/subscribe logic that you can use as the foundation for a chat application. If you run the following script you'll have a simple chat server.

Listing 3.7 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;                                    (1)
  this.subscriptions[id] = function(senderId, message) {
    if (id != senderId) {                                       (2)
      this.clients[id].write(message);
    }
  }
  this.on('broadcast', this.subscriptions[id]);                 (3)
});

var server = net.createServer(function (client) {
  var id = client.remoteAddress + ':' + client.remotePort;
  client.on('connect', function() {
    channel.emit('join', id, client);                           (4)
  });
  client.on('data', function(data) {
    data = data.toString();
    channel.emit('broadcast', id, data);                        (5)
  });
});
server.listen(8888);
```

(1) Add a listener for the "join" event that stores user's client object so the application can send data back to the user.
(2) Ignore data if it's been broadcast by the user herself.
(3) Add a listener, specific to the current user, for the "broadcast" event.
(4) Emit a "join" event when a user connects to the server, specifying the user ID and client object.
(5) Emit channel broadcast event, specifying the user ID and message, when any user sends data.

Once you've got the chat server running, open a new command-line and enter the following 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 anyone who closes their

connection, leaving the chat room, leaves 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, we add the following listener to the channel event emitter and added 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.

```
...
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);
```

If, for whatever reason, you wanted to prevent any chat without shutting 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');
});
```

SIDEBAR

Error Handling

A convention in the creation of 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.

Following is an example of an error listener handling an emitted error by logging to 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 the point where 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 behaviour 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. You can define your own response to this error type, however, by defining a global handler using the following code.

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

If you wanted 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");
  ...
```

If you wanted to increase the number of listeners an event emitter has, to avoid warnings Node will display once there are more than 10 listeners, you could use the `setMaxListeners` method. Using our channel event emitter as an example, you'd use the following line 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. Let's create, as an example of this, a class called "Watcher" meant to 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 these files into a separate directory.

The first thing you'd do is create a class contructor, as shown by the following code, that takes as arguments the directory to monitor and the directory in which to put altered files.

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

Next, you'd add logic to inherit the event emitter's behaviour.

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

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

Next, you'd extend the methods inherited from EventEmitter, as show in listing 3.8, with two new methods.

Listing 3.8 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(index in files) {
      watcher.emit('process', files[index]);
    }
  })
}

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

1. **Extend EventEmitter with method that processes files**
2. **Process each file in the watch directory**
3. **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.

```
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.

```
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 the 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 to create new classes from.

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.

3.3 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 only happen before or after other tasks.

The concept of sequencing groups of asynchronous tasks is called "flow control" by the Node community. There are two types of flow control, as figure 3.5

shows: "serial" and "parallel".

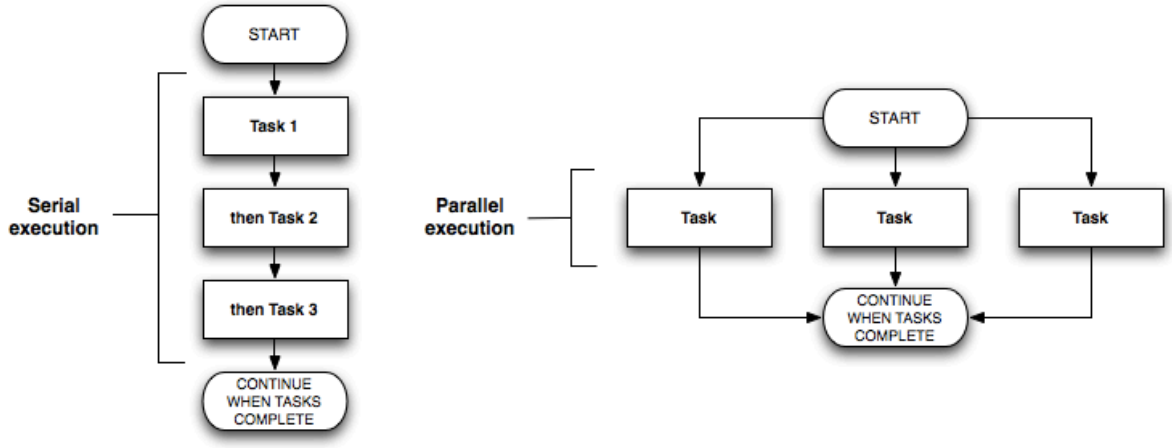

Figure 3.5 Serial execution of asynchronous tasks is similar, conceptually, to synchronous logic: tasks are executed in sequence. Parallel tasks, however, don't have to execute one after another.

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 downloads should be completed before moving on to creating the archive.

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

Flow control tools handle the bookkeeping for you, making 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 both how to implement flow control yourself or with community-created tools. To start, we're going to look at serial control flow.

3.3.1 When to use serial control flow

In order to execute a number of asynchronous tasks in sequence, you could use callbacks, but if there are a significant number of tasks you'd have to make an effort to organize them or you'd end up with messy code due to execessive callback nesting.

The following code is an example of executing tasks in sequence using callbacks. In our example we use `setTimeout` to simulate tasks that take some time to execute: the first task takes one second, the next takes half 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 programatically 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);
```

Following is an example of using nimble, a community-created flow control tool, to execute these tasks using serial control flow.

Listing 3.9 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);
  }
]);
```

While the implementation using control flow is 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.

3.3.2 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 order of desired execution. This array, as figure 3.6 shows, will act as a queue: when you finish one task you extract the next task in sequence from the array.

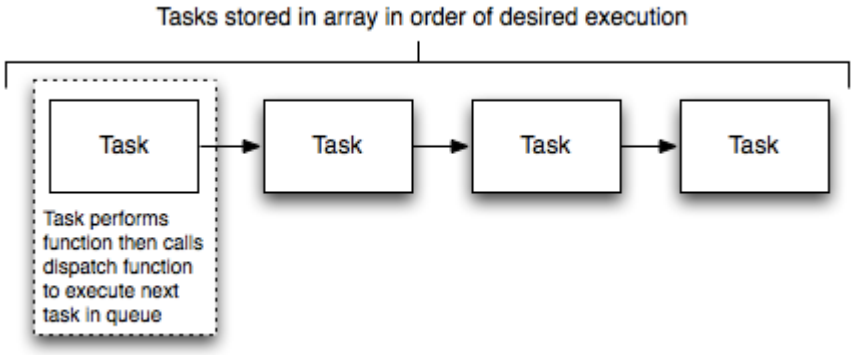

Figure 3.6 A visual representation of how serial control flow works.

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. 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, containing the following code.

Listing 3.10 random_story.js: serial control flow implemented in a simple application

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

var tasks = [                                                    ❶
  function() {
    var configFilename = './rss_feeds.txt';
    path.exists(configFilename, function(exists) {               ❷
      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) {        ❸
      if (err) {
        next(err.message);
      } else {
```

```
        feedList = feedList                                          4
        .toString()
        .replace(/^\s+|\s+$/g, '')
        .split("\n");
        var random = Math.floor(Math.random() * feedList.length);    5
        next(false, feedList[random]);
      }
    });
  },
  function(feedUrl) {
    request({uri: feedUrl}, function(err, response, body) {          6
      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);                                       7

    if (handler.dom.items.length) {
      var item = handler.dom.items.shift();
      console.log(item.title);                                       8
      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) {
    currentTask(result);                                             12
  }
}

next();                                                              13
```

1. 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.
2. The first task is to make sure the file containing the list of RSS feed URLs, "rss_feeds.txt", exists.
3.

3. The second task starts by reading a file containing the feed URLs.
4. 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.
5. The second task completes with the selection of a random feed URL from the array of feed URLs.
6. The third task uses the third-party "request" module to do an HTTP request for the randomly selected feed.
7. The fourth task starts by parsing the RSS data, from the previous HTTP request, into array of items.
8. The fourth task continues by displaying the title and URL of the first feed item, if it exists.
9. A function called "next" handles the execution of each task.
10. If a task encounters an error, an exception is thrown.
11. Get the next task from the array of tasks.
12. Execute the current task.
13. 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, lets look at how to execute asynchronous tasks in parallel.

3.3.3 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. How this

application words is shown visually in figure 3.7.

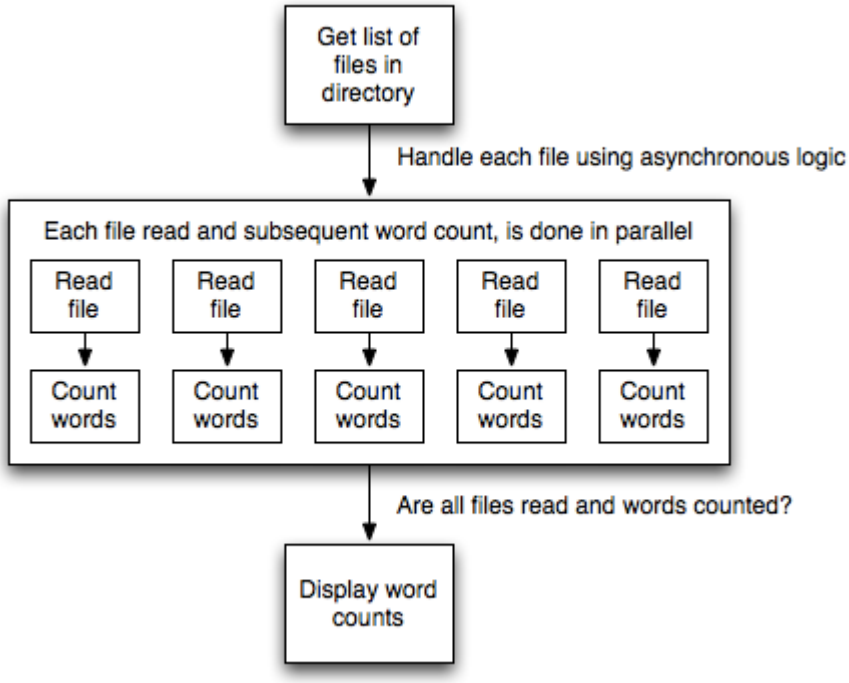

Figure 3.7 A visual representation of using parallel control flow to implement a count of word use in a number of files.

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, containing the following code.

Listing 3.11 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) {                                    (1)
      console.log(index +': ' + wordCounts[index]);
    }
  }
}

function countWordsInText(text) {
  var words = text
    .toString()
    .toLowerCase()
    .split(/\W+/)
    .sort();
  for(var index in words) {                                           (2)
    var word = words[index];
    if (word) {
      wordCounts[word] = (wordCounts[word]) ? wordCounts[word] + 1 : 1;
    }
  }
}

fs.readdir(filesDir, function(err, files) {                           (3)
  if (err) throw err;
  for(var index in files) {
    var task = (function(file) {                                      (4)
      return function() {
        fs.readFile(file, function(err, text) {
          if (err) throw err;
          countWordsInText(text);
          checkIfComplete();
        });
      }
    })(filesDir + '/' + files[index]);
    tasks.push(task);                                                 (5)
  }
  for(var task in tasks) {                                            (6)
    tasks[task]();
  }
});
```

(1)

When all tasks have completed, displays a list of each word used in the files and how many times it was used.
2. Count word occurrances in text
3. Gets a list of the files in the "text" directory.
4. Define a task to handle each file. Each task includes a call to a function that will count the file's word usage.
5. Add each task to an array of functions to call in parallel.
6. Start execution of every task.

Before trying out the application, create the 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 now learn how to leverage community-created tools that allow you to easily benefit from control flow, in your applications, without having to implement it yourself.

3.3.4 Leveraging community tools

Many community add-ons exist that provide convenient flow control tools. Popular add-ons are nimble, step, and seq. Although they're all worth checking out, we'll show another example using nimble as it's very straightforward and stands out by having the smallest codebase (a mere 837 bytes, minified and compressed).

Following is an example of using nimble to sequence tasks in a script that downloads two files simultaneously, using parallel flow control, then archives them. We use serial control to make sure that the downloading is done before proceeding to archiving.

Listing 3.12 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';
```

1 **Download Node source code for a given version**

```
  exec('wget ' + url + ' -O ' + filepath, callback);
}

flow.series([
  function (callback) {
    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();
      }
    );
  }
]);
```

2 Execute a series of tasks in sequence

3 Execute downloads in parallel

4 Create archive flile

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.

3.4 Summary

In this chapter, you've learned how to deal with the challenges of controlling asynchronous behavior through the use of 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.

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.

4 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

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 favours simplicity, and high-level "sugar" APIs are left for 3rd-party frameworks such as Connect or Express, which 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.

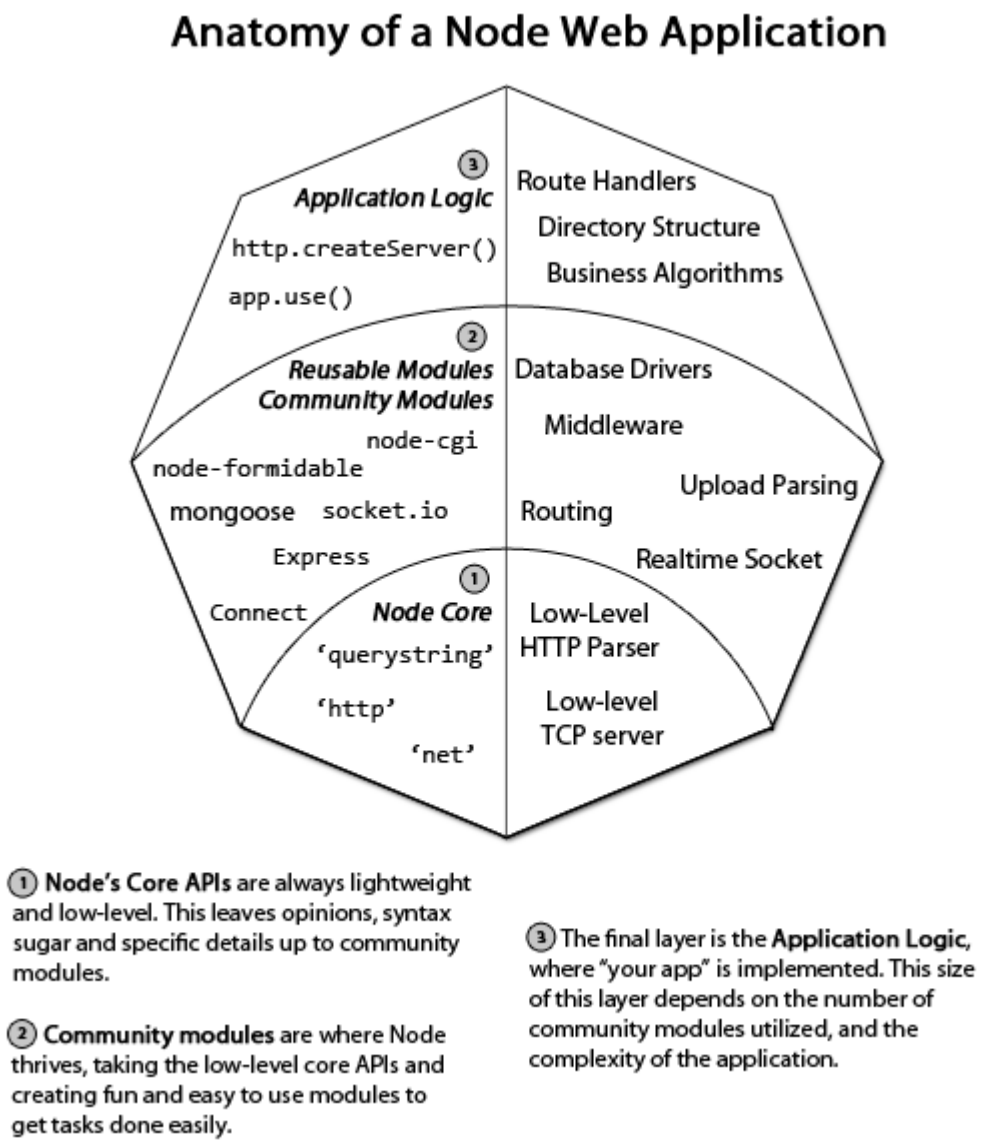

Figure 4.1 Overview of the layers that make up a Node web application

In this chapter you will become familiar with the tools Node provides us to create HTTP servers, as well as get acquainted with the 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.

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 Chapters 7 and 8 respectively. 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 elegant 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 single argument, a callback function, which will be the "request" handler used for each HTTP request received by the server. The callback accepts the `request` and `response` objects, which are commonly shortened to `req` and `res`.

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.

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

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

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 only up the end of

the 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.

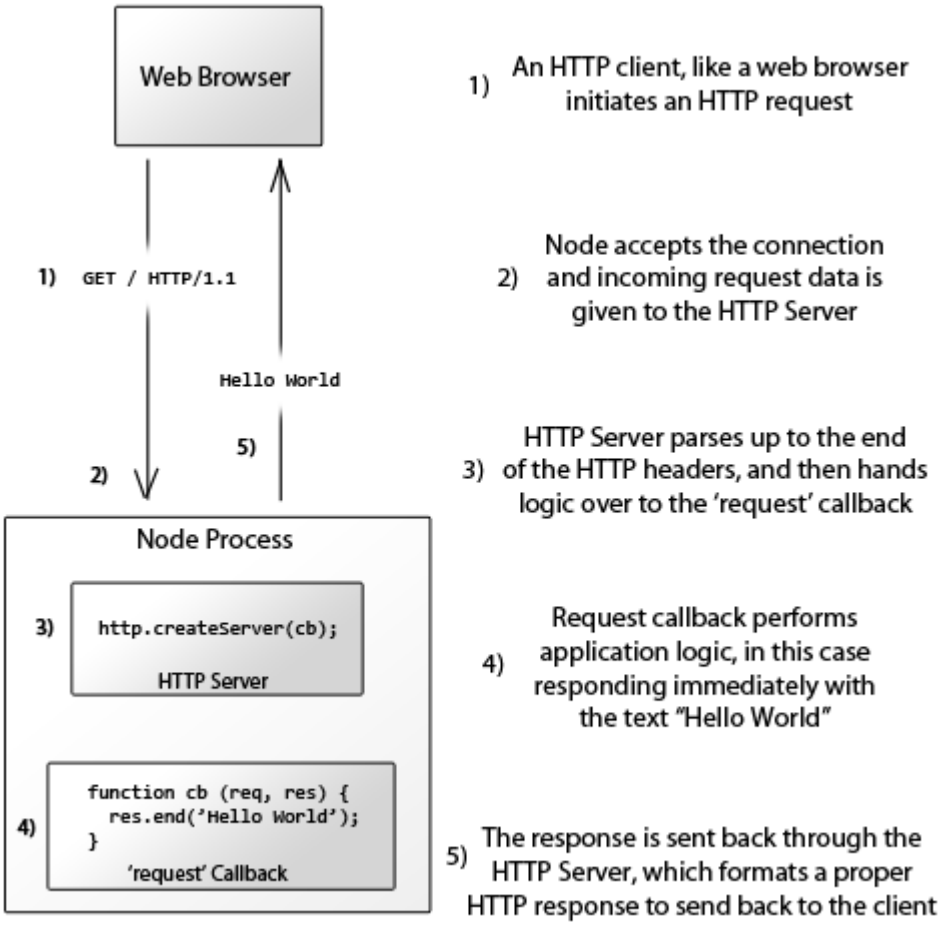

Figure 4.2 The lifecycle of an HTTP request going through a Node HTTP server

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`), for local use.

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

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

server.listen(3000, '127.0.0.1');
```

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".

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, note that these same operations with Node always reference the response object, as it may handle many requests concurrently.

```
<?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'
  , 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 Ruby On 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, and are left to other frameworks or 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` 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:

- The `POST` verb will add items to the TODO list.
- The `GET` verb will display a listing of the current items.
- 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 occured, 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. The use of `res.end('OK\n')` in listing 4.1 is functionally equivalent to a `res.write('OK\n')` call followed by a call to `res.end()`.

Listing 4.1 todo.js: POST request body string buffering

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

var items = [];                                                    ❶

var server = http.createServer(function(req, res){
  switch (req.method) {                                            ❷
    case 'POST':
      var item = '';                                               ❸
      req.setEncoding('utf8');                                     ❹
      req.on('data', function(chunk){
        item += chunk;                                             ❺
      });
      req.on('end', function(){
        items.push(item);                                          ❻
        res.end('OK\n');
```

```
      });
      break;
  }
});
```

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.

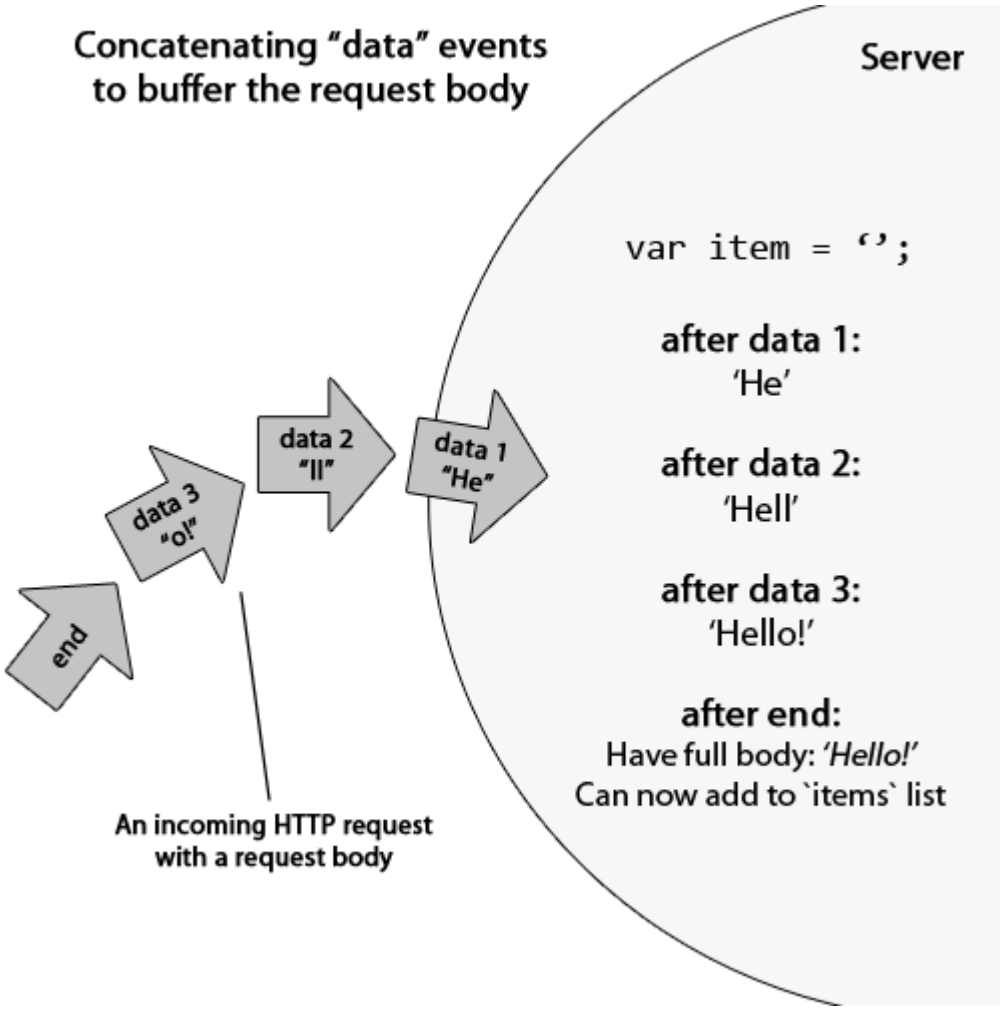

Figure 4.3 Concatenating "data" events to buffer the request body

Now the application can add items, but before trying it out using `curl`, you should complete the next task so you an get a listing 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 enitre 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 transfered. 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 would probably 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 dilema, Node gives us the `Buffer.byteLength()` method.

The following Node REPL session 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
    , 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;
...
```

1. **Add the DELETE case to the switch statement**
2. **Check that the number is valid**
3. **Ensure the requested index exists**
4. **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 statc 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')
  , parse = require('url').parse
  , join = require('path').join
  , 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')
  , parse = require('url').parse
  , join = require('path').join
  , 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);
```

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')
  , parse = require('url').parse
  , join = require('path').join
  , fs = require('fs');

var root = __dirname;

var server = http.createServer(function(req, res){
  var url = parse(req.url);
  var path = join(root, url.pathname);                    (1)
  var stream = fs.createReadStream(path);                 (2)
  stream.on('data', function(chunk){                      (3)
    res.write(chunk);
  });
  stream.on('end', function(){
    res.end();                                            (4)
```

```
  });
});

server.listen(3000);
```

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);
});
```

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()`.

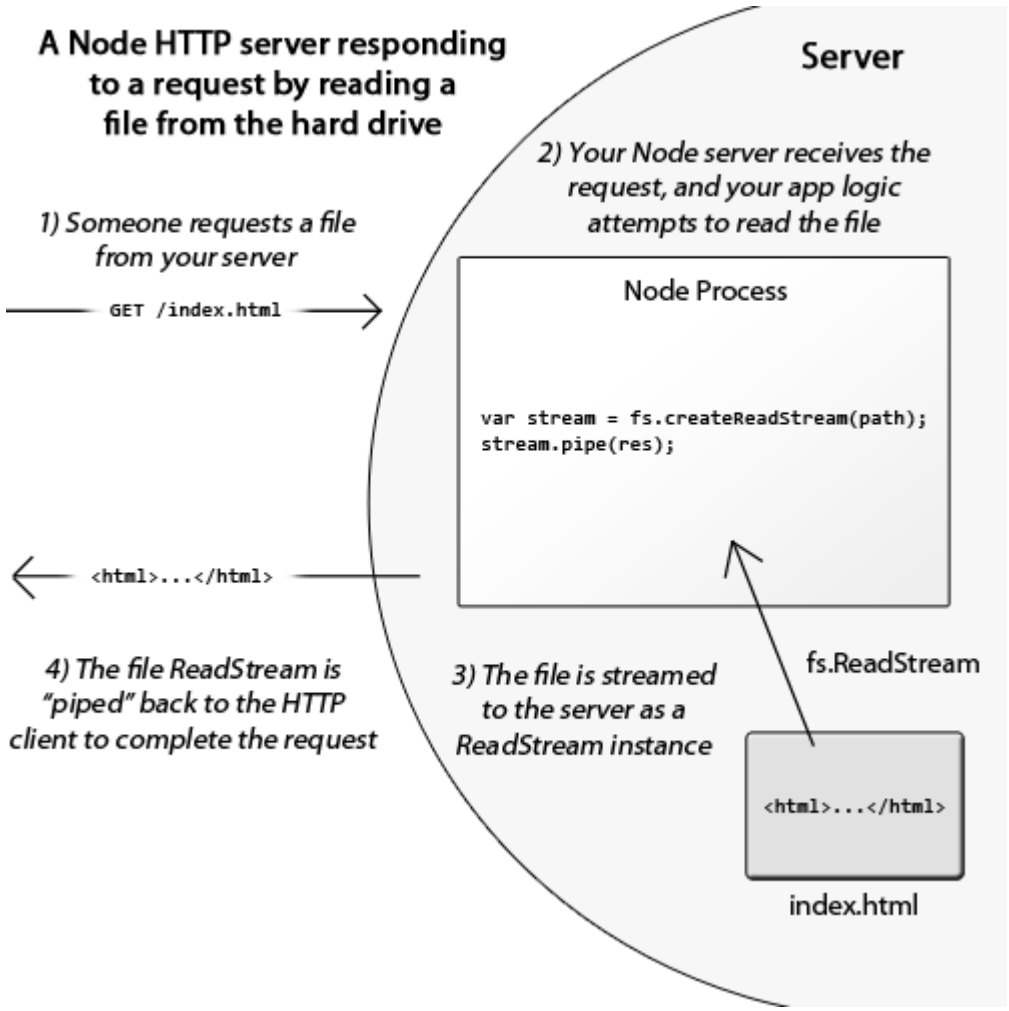

Figure 4.4 A HTTP server serving a static file from the filesystem using fs.ReadStream

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

(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 is possibly the most common form of 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, and "multipart/form-data" when containing files, non-ascii or binary data. In this section you will re-write your todo list from the previous section to utilize an html form and a web browser. When you're done, you'll have a web-based todo list that looks like the one in figure 4.5.

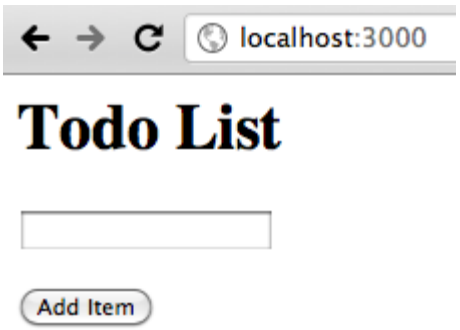

Figure 4.5 A todo-list application utilizing an HTML form and a web browser

To get started, a `switch` on the request method, or `req.method` is used in listing 4.5 to form a primitive router. Any url that is not *exactly* "/" is considered 404 "Not Found", and any HTTP verb that is not GET or POST is 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>'
    + '<form method="post" action="/">'
    + '<p><input type="text" name="item" /></p>'
    + '<p><input type="submit" value="Add Item" /></p>'
    + '</form>';
```

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

```
  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.

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.

NODE'S "QUERYSTRING" MODULE

In the server's `add()` function implementation you utilize 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 "/".

```
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);
  });
}
```

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).

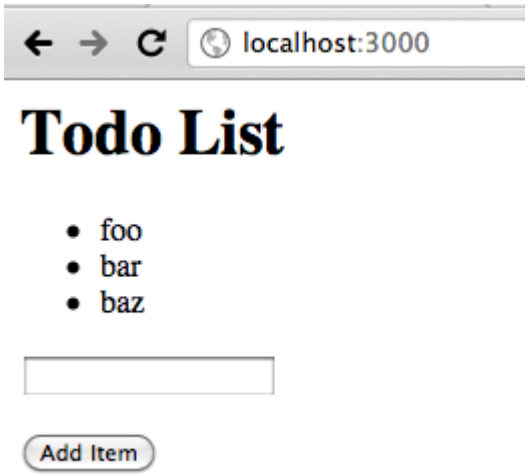

Figure 4.6 The Node todo list application in action

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 create an app where you can upload your photo collection so that you can share them to others with 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 Geisendörfer 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 500mb 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) {                                          (1)
  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) {
  // upload logic
}
```

(1) Serve an HTML form with a file input

Now that the `GET` request case it 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's needed in order to integrate "node-formidable" into your web application in order to parse file uploads:

- 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".
- 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')
  , 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()`,

providing the object with the target request so that it may 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(name);
  console.log(value);
});

form.on('file', function(name, file){
  console.log(field);
  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, it's path in the `incomingForm.uploadDir` directory ("/tmp" by default), original basename, and MIME type. The `file`

object may 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 written to stdout 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. Leave yourself a note to come back here after reading section 12.1, Socket.io, and implementing upload progress to the users should be easy.

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 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 can't be easily downloaded, or otherwise accessed, by untrusted users. In this section you will generate what is called a "self signed certificate". These kind of SSL certificates can not 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'll need OpenSSL, which will already be installed on your system if you installed Node using the instructions in Chapter 2. 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')
  , fs = require('fs');

var options = {
    key:  fs.readFileSync('./key.pem')                    (1)
  , cert: fs.readFileSync('./key-cert.pem')               (2)
};

                                                          (3)
https.createServer(options, function (req, res) {
  res.writeHead(200);                                     (4)
  res.end("hello world\n");                               (5)
}).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://127.0.0.1:3000/` 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 CA [1] and get a real, trusted SSL certificate for use with your server.

Footnote 1 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 response 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 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.

5 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 the data will need to be read and written to maintain performance, how much of it will exist, how the data will need to be queried, and persistance: how long and reliably the data needs to be stored. Methods of storing data range from stashing data in server memory to interfacing with a full-blown database management system (DBMS), but all methods require tradeoffs of one sort or another.

Some data is relatively simple -- log entries for example -- but some data is more complex, involving different types of data and relations between them. To make data complex, persistant, and fully queryable, as with a traditional DBMS, you incur a performance cost so it's not always the best strategy. Storing data in server memory, on the other hand, is performant, but less reliably persistant 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, its not unusal to use different storage mechanisms for different use-cases. In this chapter we'll talk about both how to store data without having to install and configure a DBMS and how to store

data using MySQL, PostgreSQL, Redis, and MongoDB. You'll use some of these storage mechanisms to build applications later in the book. By the end of the chapter you'll be able to leverage these varied storage mechanisms to address your application needs.

First up though, let's look at the easiest and lowest level of storage possible: serverless data storage.

5.1 Serverless data storage

The most convenient storage mechanisms, from the standpoint of systems administration, are mechanisms that don't require you to maintain a DBMS to use them: 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. Because of this, it's a perfect fit for applications you intend others to run on their own hardware. A Node-driven command-line tool for analyzing the metadata of a photographs in a directory, for example, may require data storage, but the end-user likely won't want to have to go through the hassle of setting up a MySQL server in order to use it.

In addition to CLI tools, web applications may be created that are intended to be installed and run locally. Two examples of this type of application are TJ Holowaychuk's Node-driven "Finance"[1] application, for tracking personal finances, and Google's Java-driven "Refine"[2] application, shown in figure 5.1, for cleaning up irregular datasets.

Footnote 1 https://github.com/visionmedia/finance

Footnote 2 http://code.google.com/p/google-refine/

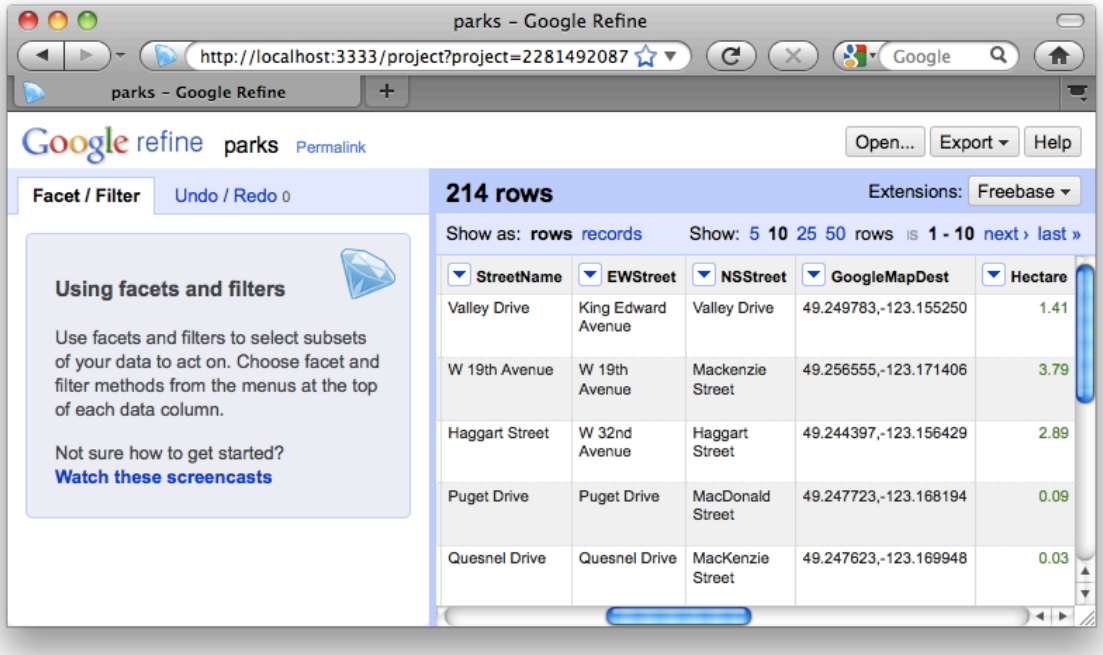

Figure 5.1 Google Refine is an example of a web application that is installed and run locally by non-developer users, thus a perfect fit for serverless data storage.

In this 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. You'll also learn about an advanced form of file-based storage, SQLite. First, let's start with the simplest of the two: in-memory storage.

5.1.1 In-memory storage

In-memory storage is the use of server RAM to store data. Reading and writing this data is fast, but data, as mentioned earlier, is lost during server and application restarts.

The ideal use of in-memory storage is for small bits of frequently accessed data. One such use 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 will count 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 will persist beyond application and server restarts, file-based storage may be suitable.

5.1.2 File-based storage

File-based storage is the use of a server's filesystem to store data. It's often used to store application configuration information but is also easy way of persisting data that will survive application and server restarts.

To illustrate the use of file-based storage, let's create a simple command-line Node application for storing tasks. The application will store tasks in a file named ".tasks" in whatever directory the script is run. Tasks will be converted to JSON before being stored and will be converted from JSON when being read from the file.

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.

```
var fs = require('fs')
  , path = require('path')
  , args = process.argv.splice(2)
  , command = args.shift()
  , taskDescription = args.join(' ')
  , file = path.join(process.cwd(), '/.tasks');
```

The application then, if a command is provided, either outputs a list of stored tasks or adds a task description to the task store as shown in listing 5.1. If no command is given, the command usage is displayed.

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]');
}
```

Next, define in the application logic a helper function, `getTasks`, is defined to retrieve existing tasks. As listing 5.2 shows, `getTasks` loads in a text file in which JSON-encoded data is stored.

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

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

1. **Make sure todo file exists**
2. **Read todo data from file**
3. **Parse JSON-encoded todo data**

Then, use the `getTasks` helper function to implement the `listTasks` functionality.

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

Next, define in the application logic another helper function, `storeTasks`, to store tasks.

```
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) {
  getTasks(file, function(tasks) {
    tasks.push(taskDescription);
    storeTasks(file, tasks);
  });
}
```

Using the filesystem as a data store is a relatively quick and easy way to add persistance 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 `getTasks` could be repurposed to read the file and parse the JSON.

In chapter 11 "Beyond Web Servers" you'll learn more about manipulating the filesystem with Node. In the meantime, let's move on to an alternate approach to store data in the filesystem: SQLite.

5.1.3 SQLite

Let's say you're creating a serverless web application for keeping 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 data store, 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.

A better way to handle data storage in this application would be to use a data store that you could query, such as a relational database management system (RDBMS). Most RDBMSs require a server application be configured and installed but SQLite is an exception.

SQLite is a relational database designed to simplify the setup of applications that don't need all the features of a full-blown DBMS. As such it doesn't require the installation of additional database server software and requires no specialized administration knowledge. SQLite does, however, require a working knowledge of Structured Query Language (SQL), which is outside of the scope of this book. For those new to SQL, there are many online tutorials and books, including Chris Fehily's "SQL: Visual QuickStart Guide", that can help you get up to speed.

There are two SQLite API modules currently being actively developed: Development Seed's `node-sqlite3`[3] and Orlando Vazquez's `node-sqlite`[4]. Development Seed's module builds upon Vazquez's module and has more features, such as integrated flow control to sequence queries. Vazquez's module, in turn, has been around the longest and is the most widely used. As Vazquez's module has the simplest API we'll use it for our examples.

Footnote 3 https://github.com/developmentseed/node-sqlite3

Footnote 4 https://github.com/orlandov/node-sqlite

Note that for installing many Node database interface modules, your workstation or server will require the same development tools that are used to compile Node from source code. See chapter 2, sections 2.2.1-2.2.3, for operating-system specific instructions.

Install `node-sqlite` using the following command.

```
npm install sqlite
```

As an example of how you could use SQLite for an application, we're going to go through how you'd build a serverless work tracking web application with it. Our application will be composed of two files: `timetrack_server.js`, used to start the application, and `timetrack.js`, application 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 SQLite API.

```
var http = require('http')
  , work = require('./lib/timetrack')
```

```
  , sqlite = require('sqlite')
  , db = new sqlite.Database();
```

Next, add the following logic to `timetrack_server.js` to define the basic web application behavior. The application allows you to browse, add, and delete records of work performed. In addition, work records can be archived. Archiving a work record hides it on the main page. Archived records, however, 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':
          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;
  }
});
```

1 **Route HTTP POST requests**

2 **Route HTTP GET requests**

The following code is the final addition to `timetrack_server.js`. This logic opens the SQLite database, creates a database table if none exists, and starts the HTTP server. All node-sqlite queries are performed using the `execute` function.

Listing 5.4 timetrack_server.js: Database table creation

```
db.open('timetrack.sqlite', function(err) {
  if (err) throw err;
  db.execute(
    "CREATE TABLE IF NOT EXISTS work ("
    + "id INTEGER PRIMARY KEY, "
    + "hours REAL DEFAULT 0, "
    + "date TEXT, "
    + "archived INTEGER DEFAULT 0, "
    + "description TEXT)",
    function(err) {
      if (err) throw err;
      console.log('Server started...');
      server.listen(3000, '127.0.0.1');
    }
  );
});
```

1 **Open SQLite database**

2 **Table creation SQL**

3 **Start HTTP server**

Now that you've fully defined the file used to start the application, lets 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 put the following logic, which includes the Node `querystring` API and defines helper functions for sending web page HTML and receiving data submitted through forms, in this file.

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.parseSentData = 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);
  });
}
```

1 **Send HTML response**

2 **Parse HTTP POST 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;
}
```

❸ **Render simple form**

Next, add the following logic to `timetrack.js` to define logic that will add a work record to the SQLite database.

The character "?" is used as a placeholder to indicate where a parameter should be placed. Each parameter is escaped before being added to the query, preventing SQL injection attacks. Note that the second argument of the `execute` 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.parseSentData(req, function(work) {
    db.execute(
      "INSERT INTO work (hours, date, description) " +
      " VALUES (?, ?, ?)",
      [work.hours, work.date, work.description],
      function(err) {
        if (err) throw err;
        exports.show(db, res);
      }
    );
  });
}
```

❶ **Parse HTTP POST data**
❷ **SQL to add work record**
❸ **Work record data**
❹ **Show user a list of work records**

Next, add the following logic 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.parseSentData(req, function(work) {
    db.execute(
      "DELETE FROM work WHERE id=?",
      [work.id],
      function(err) {
```

❶
❷
❸

```
        if (err) throw err;
        exports.show(db, res);                                    (4)
      }
    );
  });
}
```

(1) Parse HTTP POST data
(2) SQL to delete work record
(3) Work record ID
(4) Show user a list of work records

Next, add the following logic 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.parseSentData(req, function(work) {                     (1)
    db.execute(
      "UPDATE work SET archived=1 WHERE id=?",                    (2)
      [work.id],                                                  (3)
      function(err) {
        if (err) throw err;
        exports.show(db, res);                                    (4)
      }
    );
  });
}
```

(1) Parse HTTP POST data
(2) SQL to update work record
(3) Work record ID
(4) Show user a list of work records

Now that you've got logic defined that will add, delete, and update a work record, you'll next add the following logic to retrieve work record data, archived or unarchived, so it can be rendered as HTML. When issuing a query that will return data, the callback that defines what to do after the query should have a "rows" argument.

Listing 5.9 timetrack.js: Retrieving work records

```
exports.show = function(db, res, showArchived) {
  var query = "SELECT * FROM work " +                          ❶
    "WHERE archived=? " +
    "ORDER BY date DESC";
  var archiveValue = (showArchived) ? 1 : 0;
  db.execute(
    query,
    [archiveValue],                                            ❷
    function(err, rows) {
      if (err) throw err;
      html = (showArchived)
        ? ''
        : '<a href="/archived">Archived Work</a><br/>';
      html += exports.workHitlistHtml(rows);                   ❸
      html += exports.workFormHtml();
      exports.sendHtml(res, html);                             ❹
    }
  );
}

exports.showArchived = function(db, res) {
  exports.show(db, res, true);                                 ❺
}
```

❶ SQL to fetch work records
❷ Desired work record archive status
❸ Format results as HTML table
❹ Send HTML response to user
❺ Show only archived work records

Add the following logic to `timetrack.js` to do the actual 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) {                                         ❶
    html += '<tr>';
    html += '<td>' + rows[i].date + '</td>';
    html += '<td>' + rows[i].hours + '</td>';
    html += '<td>' + rows[i].description + '</td>';
    if (!rows[i].archived) {                                   ❷
      html += '<td>' + exports.workArchiveForm(rows[i].id) + '</td>';
    }
    html += '<td>' + exports.workDeleteForm(rows[i].id) + '</td>';
    html += '</tr>';
  }
```

```
  html += '</table>';
  return html;
}
```

1. Render each work record as an HTML table row
2. Show archive button if work record isn't already archived

Finally, add the following code 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="/">' +
    'Date (YYYY-MM-DD): <input name="date" type="text"><br/>' +
    'Hours worked: <input name="hours" type="text"><br/>' +
    '<p>Description:<br/>' +
    '<textarea name="description"></textarea></p>' +
    '<input type="submit" value="Add" />' +
    '</form>';
  return html;
}

exports.workArchiveForm = function(id) {
  return exports.actionForm(id, '/archive', 'Archive');
}

exports.workDeleteForm = function(id) {
  return exports.actionForm(id, '/delete', 'Delete');
}
```

1. **Render blank HTML form for entry of new work record**
2. **Render archive button form**
3. **Render delete button form**

Now that you've fully defined the application, you can start it by entering the following into your command-line then navigating to `http://127.0.0.1:3000/` in a web browser.

```
node timetrack_server.js
```

While SQLite is very useful, it isn't meant to replace full-blown DBMSs. DBMSs have many features that help with performance and reliability, such as the

ability to automatically replicate data between servers. If the performance of SQLite is insufficient for your application, or SQLite doesn't have the features you need, you'll likely want to look instead at relational database management systems.

5.2 Relational database management systems

RDBMSs allow complex information to be stored and easily queried. RDBMSs have traditionally been used for relatively complex 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 SQL, although object-relational mappers (ORMs) exist that provide APIs that will write SQL for you in the background. RDBMS administration, ORMs, and SQL are out of the scope of this book, but there are many online resources covering these technologies.

Many relational databases exist but most developers choose open-source databases, primarily because they're well-supported, work well, and don't cost anything. In this section we'll look at the two most popular fully-featured relational databases: MySQL and PostgreSQL. MySQL and PostgreSQL both have similar capabilities and either is a solid choice. If you haven't used either, MySQL is probably the easiest to start with as it's easier to set up and has a larger userbase. If you use happen to use the proprietary Oracle Database, you'll want to use the db-oracle[5] module which is also outside the scope of this book.

Footnote 5 https://github.com/mariano/node-db-oracle

Let's get started, looking first at MySQL, then at PostgreSQL.

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, there is an official tutorial available online[6].

Footnote 6 http://dev.mysql.com/doc/refman/5.0/en/tutorial.html

The most popular MySQL API module is Felix Geisendörfer's `node-mysql`[7] module. Install this module via npm using the following command.

Footnote 7 https://github.com/felixge/node-mysql

```
npm install mysql
```

To show MySQL in action, lets look at how you'd change the work tracking application created earlier to use MySQL instead of SQLite. Once you've installed the node-mysql module, you'd need to connect to MySQL, and select a database to query. To make `timetrack_server.js` connect to MySQL instead of SQLite you'd replace the first four lines of code with the following:

```
var http = require('http')
  , work = require('./lib/timetrack')
  , mysql = require('mysql');

var db = mysql.createClient({
  user:     'myuser',
  password: 'mypassword',
});

db.useDatabase('timetrack');
```

The next part of the code you'd need to modify is the table creation code. You'd remove the line in `timetrack_server.js` that begins with "db.open" and all lines that come after it. You'd then replace these lines with the following code:

Listing 5.12 timetrack_server_mysql.js: Database table creation

```
db.query(
  "CREATE TABLE IF NOT EXISTS work ("                    (1) Table creation SQL
  + "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(300, '127.0.0.1');                     (2) Start HTTP server
  }
);
```

The `query` method is used for all queries in mysql-node. The following code shows how you'd modify the SQLite application example's `lib/timetrack.js` file to insert into a MySQL database table instead. The node-mysql API is

extremely similar to the node-sqlite API, in fact, so all that's required to change it to use MySQL is to replace each call to the `execute` method in `lib/timetrack.js` to be a call to the `query` method.

Listing 5.13 timetrack_mysql.js: Adding a work record

```
exports.add = function(db, req, res) {
  exports.parseSentData(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);
      }
    );
  });
}
```

1. **Parse HTTP POST data**
2. **SQL to add work record**
3. **Work record data**
4. **Show user a list of work records**

As with SQLite's API, the character "?" is used to indicate where a parameter should be placed in the SQL query. Each parameter is escaped before being added, preventing SQL injection attacks.

Sometimes, in an application, when you insert a row of data you need to subsequently know the value of the row's primary key. You can do this by adding a callback argument to the query call as the below example shows.

```
client.query(
  "INSERT INTO users " +
  "(name, age) VALUES (?, ?)",
  ['Rico', 23],
  function(err, result) {
    if (err) throw err;
    console.log('New row ID is ' + result.insertId + '.');
  }
);
```

If you're creating a query that will return results, the final argument of the `query` method must be a callback. Listing 5.14 shows the use of a callback to output data returned from a query.

Listing 5.14 select_mysql.js: Selecting rows from a MySQL database

```
client.query(
  "SELECT * FROM users",
  function(err, results, fields) {
    if (err) throw err;

    for (var index in results) {
      console.log(results[index].name);
    }
    client.end();
  }
);
```

Note that with a MySQL connection opened, your application will not exit until you close the connection. If you were writing a script to make changes to a database and wanted it to end cleanly after running, you'd call the `end` method, as shown in the following code.

```
db.end();
```

Although MySQL is the most popular relational database, PostgreSQL is, by many, the most respected and we'll now look at how to leverage it in your application.

5.2.2 PostgreSQL

PostgreSQL is well-regarded for its standards compliance and robustness and is the favored RDBM of many Node developers. If you're new to PostgreSQL and interested in learning it, there is an official tutorial available online[8].

Footnote 8 http://www.postgresql.org/docs/7.4/static/tutorial.html

The most mature and actively developed PostgreSQL API module is Brian Carlson's node-postgres[9]. Install this module via npm using the following command.

Footnote 9 https://github.com/brianc/node-postgres

```
npm install pg
```

Once you've installed the node-postgres module, you can connect to PostgreSQL, 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();
```

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 a parameter should be placed. 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]
);
```

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, leveraging built-in Node functionality.

This object emits a 'row' event for each retrieved database row. Listing 5.15 shows how you can output data from each row returned by a query. Note the use of event emitters listeners that define what to do with database table rows and what to do when data retreival is complete.

Listing 5.15 select_postgres.js: 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.

NOTE **Insert IDs**

The node-postgres API doesn't provide an easy way to get the primary key value of the last row inserted. To do this you must separately query the `last_value` column of the sequence used to provide a table's primary key value.

Relational databases are classic workhorses, but another breed of database manager, not requiring 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. Relational databases, however, slowly gained popularity and became the mainstream choice for applications both on and off the web. In recent years, however, non-relational DBMSs have re-emerged with proponents claiming advantages in scalability and simplicity. Many of these DBMSs now exist targeted towards 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 there are instructions for Redis[10] and MongoDB[11] that should help you get up and running.

Footnote 10 http://redis.io/topics/quickstart

Footnote 11 http://www.mongodb.org/display/DOCS/Quickstart+OS+X

5.3.1 Redis

Redis is a data store well-suited to handling simple, relatively ephemeral data such as logs, votes, and messages. It provides a vocabulary of primitive, but useful, commands[12] 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: hashes, 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 further on.

Footnote 12 http://redis.io/commands

While we won't go into all of Redis's commands in this section we'll run through a number of examples that will 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 Tutorial[13].

Footnote 13 http://simonwillison.net/static/2010/redis-tutorial/

The most mature and actively developed Redis API module is Matt Ranney's `node_redis`[14] module. Install this module using the following npm command.

Footnote 14 https://github.com/mranney/node_redis

```
npm install redis
```

The following code establishes a connection to a Redis server, using the default TCP/IP port, running on the same host. The Redis client created has inherited EventEmitter behavior and emits an "error" event when the client has problems communicating with the Redis server. As shown below, 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);
});
```

Once 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:

```
client.set('color', 'red', redis.print);
var keyValue = client.get('color', function(err, value) {
  if (err) throw err;
  console.log('Got: ' + value);
});
```

Note the use, in the previous example code, of `redis.print`. `print` is a convenience function that simply outputs, using `console.log`, Redis's response and any errors encountered.

Listing 5.16 shows the storage and retrieval of values in a slightly more complicated data structure: the hash. The `hset` Redis command sets a hash element, identified by a key, to a value. The `hkeys` Redis command lists the keys of each element in a hash.

Listing 5.16 redis_keys.js Storing data in elements of a Redis hash.

```
client.hset('camping', 'shelter', '2-person tent', redis.print); (1)
client.hset('camping', 'cooking', 'campstove', redis.print);

client.hget('camping', 'cooking', function(err, value) {      (2)
  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 values
3. Get hash keys

Another data structure supported is the list. A Redis list can theoretically hold over 4 billion elements, memory permitting. The following code shows the storage and retrieval of values in a list. The `lpush` Redis commands 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 code below 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);
  });
});
```

While lists, which are similar conceptually to arrays in many programming langages, provide a familiar way to manipulate data, one downside is retrieval performance. As the list grows in length, retrieval becomes slower (O(n) in big O notation).

NOTE **Big O Notation**

Big O Notation is a way of categorizing algorithms, in computer science, by performance. Seeing an algorithim'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"[15] provides a great overview.

Footnote 15

http://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

Sets are another type of Redis data structure 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, however, 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 attempt to add the IP "204.10.37.96" twice, but when we display the set members you can see that it 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);
});
```

Redis, it's also worth noting, goes beyond the traditional role of datastore by providing "channels". Channels are a data delivery mechanism, as shown conceptually in figure 5.2, that provides "publish/subscribe" functionality, useful for chat and gaming applications.

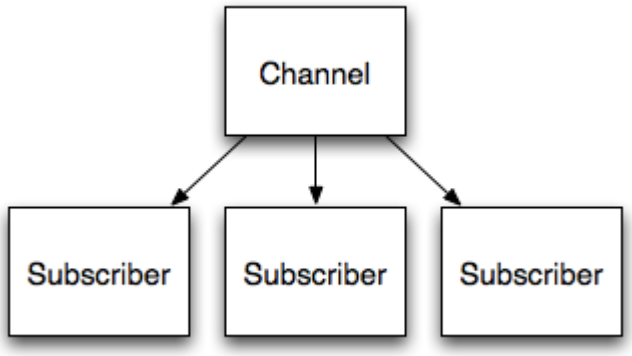

Figure 5.2 Redis channels provide an easy solution to a frequent data delivery scenario.

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 5.17 shows an example of how Redis' publish/subscribe functionality can be used to relay messages.

Listing 5.17 redis_pubsub.js: An example of Redis publish/subscribe functionality

```
var redis = require('redis')
  , clientA = redis.createClient()                          (1)
  , clientB = redis.createClient();

clientA.on('message', function(channel, message) {          (2)
  console.log('Client A got message from channel %s: %s',
    channel,
    message
  );
});

clientA.on('subscribe', function(channel, count) {          (3)
  clientB.publish('main_chat_room', 'Hello world!');        (4)
});

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

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` module[16]. Using this module will speed up Redis performance significantly because it leverages the official hiredis C library. node-redis will automatically use hiredis, if installed, instead of its JavaScript implementation. Install hiredis using the following npm command.

Footnote 16 https://github.com/pietern/hiredis-node

```
npm install hiredis
```

Note that because the hiredis library is compiled 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 sort of applications as an RDBMS and is well-regarded for its performance capabilities. It stores data in RAM before writing so it's a good choice if you wish to sacrifice reliability for speed. If reliability or availability are concerns, MongoDB has powerful replication features you can harness.

A MongoDB database stores documents in "collections". Documents in a collection, as shown in figure 5.3, need not share the same schema: each document could conceivably have a different schema. This makes MongoDB much more flexible than conventional RDBMSs as you don't have to worry about predefining schema.

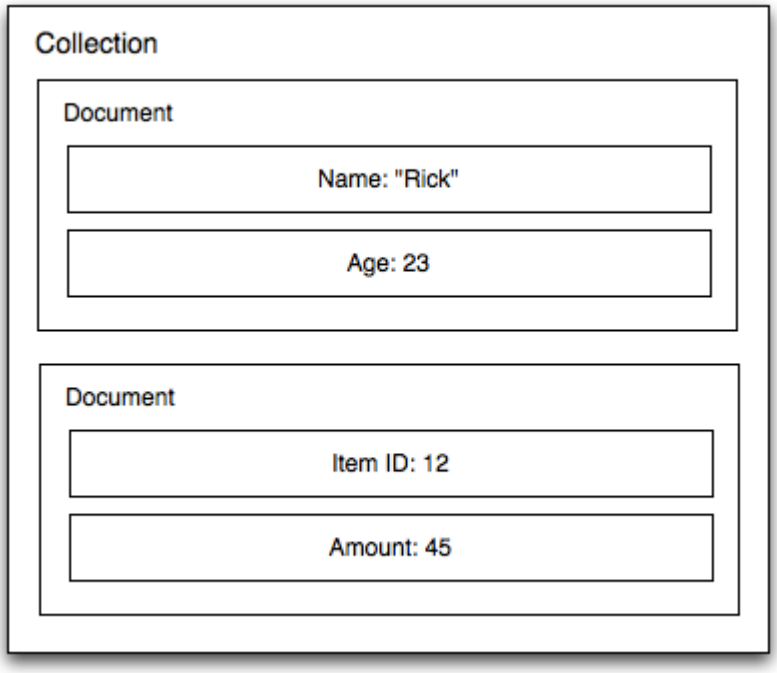

Figure 5.3 If you use MongoDB to store data for your Node

application, then 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-native[17]. Install this module using the following npm command.

Footnote 17 https://github.com/christkv/node-mongodb-native

```
npm install mongodb
```

After installing node-mongodb-native and running your MongoDB server, use the the following 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);
```

Listing 5.18 shows how you can access a collection once the database connection is open.

Listing 5.18 mongo_connect.js Connecting to a MongoDB collection

```
client.open(function(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
  });
});
```

If, at any time after completing your database operations, you want to close your MongoDB connection simply execute `client.close()`.

The code below inserts a document in a collection and prints its unique document ID.

```
collection.insert({a: 2}, {safe: true}, function(err, documents) {
  if (err) throw err;
  console.log('Document ID is: ' + documents[0]._id);
});
```

Document IDs may be used to update data. Listing 5.19 shows how to update a document using its ID.

Listing 5.19 mongo_update.js: Updating a MongoDB document

```
var _id = new client.bson_serializer
                    .ObjectID('4e650d344ac74b5a01000001');
collection.update(
  {_id: _id},
  {$set: {a: 3}},
  {safe: true},
  function(err) {
    if (err) throw err;
  }
);
```

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 the callback is executed. 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 in MongoDB is done using the `find` method. The example below shows logic that will display all items in a collection.

```
collection.find({a: 2}).toArray(function(err, results) {
  if (err) throw err;
  console.log(results);
});
```

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;
});
```

While MongoDB is a powerful database and node-mongodb-native offers high-performance access to it, you may want to use an API that abstracts database access, handling the details for you in the background, so you can develop faster and have to maintain 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 things 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. While we'll focus solely on the basic use of Mongoose use as a data store, if you decide to use Mongoose in your application you'll definitely benefit from reading its online documentation[18] and learning all it has to offer.

Footnote 18 http://mongoosejs.com/

Install Mongoose via npm using the following command.

```
npm install mongoose
```

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 wish to terminate your Mongoose-created connection, the following code will close it.

```
mongoose.disconnect();
```

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 documentation[19].

Footnote 19 http://mongoosejs.com/docs/schematypes.html

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 with Mongoose is similiarly easy. The task model's `find` method allows us to find all, or select, documents using a JavaScript object to specify our 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);
  }
});
```

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 5.20 shows how you can update a document using Mongoose.

Listing 5.20 mongoose_update.js 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;
    console.log('Updated.');
  }
);
```

1. **Update using internal ID**
2. **Only update one document**

Using Mongoose, it's easy to remove a document 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();
});
```

There is 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 traditonally associated with relational database management systems.

5.4 Summary

Having gained an understanding of a healthy selection of data storage technologies, you now have the basic knowledge needed 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 RDBMSs. 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 have 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. If you do need complex query abilities, however, SQLite takes file-based storage to the next level.

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 SQLite, 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 persistance under your belt, you've learned the fundamentals needed to make simple web applications. You're now ready to move on to testing, an important tool needed to ensure that what you code today works tomorrow.

6 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 the boring stuff for you so you can 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 so you can write tests while you develop.

In this section we'll look at two types of automated testing: unit testing and acceptance testing. Unit testing tests code logic directly and is applicable to all types of applications. Acceptance testing, however, 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 each -- for unit testing, Node's assert module and the Expresso, nodeunit, Vows, and Should.js frameworks; for acceptance testing, the Tobi and Soda frameworks.

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. By doing so you're able to quickly and easily test any changes made to your application, confident that your changes haven't introduced errors. Writing unit tests takes a bit of work up front, but they can end up saving you time by lessening the need to manually retest every time you make a change to an application.

Unit testing can be tricky, however, and asynchronous logic adds new challenges. As asynchronous unit tests can run in parallel you've got to be careful that what one test does doesn't interfere with another. For example, if your tests create temporary files on disk you'll have to be careful, when deleting the files after a test, that you don't delete the working files of another test that hasn't yet finished.

When writing unit tests, the ideal is to write tests that verify the proper functionality of all application logic. An application that has tests for most of its functionality is said to have "good code coverage". 100% coverage is the goal of testing. You likely won't reach that level with every application, but the more coverage the better.

Within the world of unit testing, you'll also find two methodological flavors: test driven development (TDD) and behavior driven development (BDD). TDD, as shown in figure 6.1, is a programming methodology that puts the focus on testing rather than the application. Adherents to TDD write unit tests first, then write application logic. Adherents of TDD swear this reduces application development time, but there are also plenty of developers who disagree. TDD isn't everyone's cup of tea, but it definitely doesn't hurt to experiment with it.

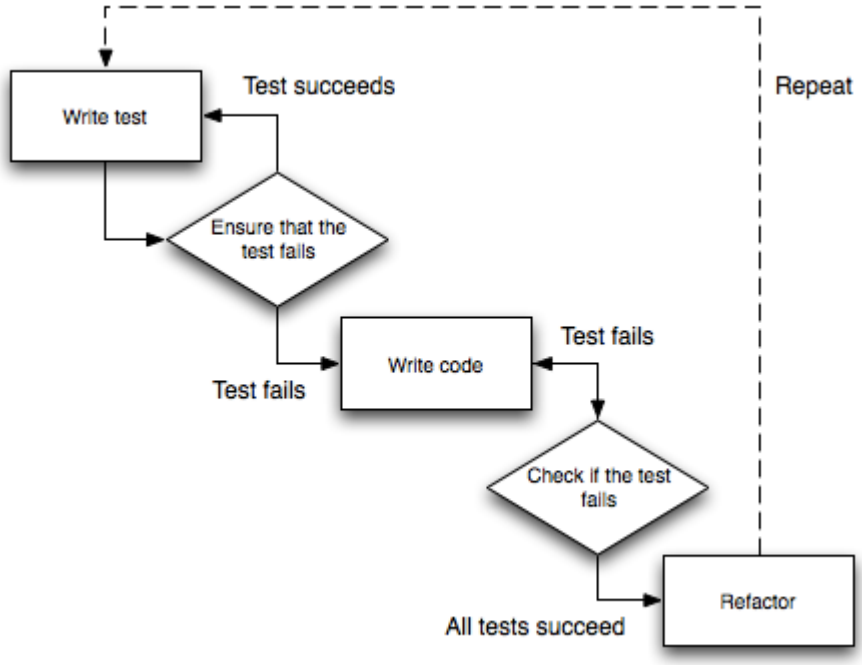

Figure 6.1 With test driven development, tests are written first, then the corresponding code is written.

Behavior driven development (BDD) tweaks the TDD philosophy and attempts to make tests easier to understand by changing the style in which tests are specified in code. The aim of BDD is to make the language used when programming tests similar to plain English. As a result, application behavior can then be vetted by all stakeholders to avoid misunderstandings about functionality.

The following assertion, for example, would be hard to understand for non-programmers.

```
assert.equal(total, 42);
```

Using the should.js BDD-flavored assertion module, however, it can be rewritten in a way that can be understood by non-programmers.

```
total.should.equal(42);
```

In this section we'll look at the Expresso and nodeunit unit testing frameworks. We'll also show how to use BDD-flavored testing tools: the Vows testing framework, that aims to make test structures easier to understand, and should.js, a module that allows you to use BDD-flavored assertions with any unit testing framework.

Before we get into those frameworks, though, lets look at unit testing with Node's built-in assert module.

6.1.1 The Assert Module

The basis of most Node unit testing is Node's built-in `assert` command. The `assert` command tests a condition and, if the condition isn't met, throws an error. Node's `assert` is leveraged by many third-party testing frameworks, but even without a testing framework you can accomplish basic testing using it.

Suppose you want to take the todo application created in chapter 4 and make it store items in a database. You'll want to rework it, adding core logic to manage storage of todos, and you'll want to add tests to make sure the core logic still works whenever you modify the application.

The following defined todo application logic uses Mongoose, which we talked about in Chapter 5, to store todo items in MongoDB. If you'd like to run the code, you'll need to run a MongoDB server[1].

Footnote 1 http://www.mongodb.org/display/DOCS/Quickstart

First, create a working directory for the application and its modules and install Mongoose.

```
mkdir -p ~/tmp/todo
cd ~/tmp/todo
mkdir lib
npm install mongoose
```

The following code defines a module containing the core application functionality. Put this file into the `lib` directory in a file called `todo.js`.

Listing 6.1 todo.js

```
var mongoose = require('mongoose')
  , db = mongoose.connect('mongodb://localhost/todos')
  , Schema = mongoose.Schema
  , Todos = new Schema({item: String})
  , Todo = mongoose.model('Todo', Todos);

exports.TodoDb = function() {};

exports.TodoDb.prototype = {

  add: function(todoData, cb) {
    var todo = new Todo();
    todo.item = todoData.item;
    todo.save(function(err) {
      cb(err, todo);
    });
```

1 **Define todo item schema**

2 **Add a todo item**

```
  },

  get: function(options, cb) {

    options.skip  = options.skip || 0;
    options.limit = options.limit || 25;

    var todos = [];
    Todo
    .find({})
    .skip(options.skip)
    .limit(options.limit)
    .exec(function(err, todos) {
      cb(err, todos);
    });
  },

  delete: function(cb) {
    this.get({}, function(err, todos) {
      for(var index in todos) {
        todos[index].remove();
      }
      cb();
    });
  },

  close: function() {
    db.disconnect();
  },

  unfinished: function(cb) {
    this.get({}, function(err, todos) {
      cb(todos.length > 0);
    });
  }
}
```

3 **Get all todo items**

4 **Delete all todo items**

5 **See if todo items exist**

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 and set some variables that will track testing progress.

```
var assert = require('assert')
  , todo = require('./lib/todo')
  , todoDb = new todo.TodoDb()
  , testsStarted = 0
  , testsCompleted = 0;
```

Next add a test of the todo application's delete functionality.

Listing 6.2 test.js Test to make sure that no todo items remain after deletion.

```
function deleteTest(cb) {
  testsStarted++;                                                    (1)
  todoDb.delete(function() {                                         (2)
    todoDb.get({'skip': 0, 'limit': 25}, function(err, todos) {      (3)
      if (err) throw err;
      assert.equal(                                                  (4)
        todos.length,
        0,
        'Failure: there should be no todos after deleting them'
      );
      testsCompleted++;                                              (5)
      cb();
    });
  });
}
```

(1) Note that test has started
(2) Delete any existing items
(3) Get items
(4) Assert that no items exist
(5) Notes that test has completed

The assert module's most used assertion is "equal". Listing 6.2 uses "equal" to test the contents of a variable. In the example a global variable is used to note that a test has started, all todo items are deleted, then application logic is used to attempt to fetch any todos. As there should be no todos, the value of `todos.length` should be `0` if the application logic is working properly. If there is a problem, an exception is thrown. If the variable `todos.length` isn't set to `0` the assertion would result in a stack trace showing an error message "Failure: there should be no todos after deleting them". After the assertion a global variable is used to note that a test has completed.

Next add a test of the todo application's add functionality.

Listing 6.3 test.js Test to make sure adding a todo works.

```
function addTest(cb) {
  testsStarted++;                                                    (1)
  todoDb.delete(function() {                                         (2)
    todoDb.add({'item': 'feed bobcat'}, function(err, todo) {        (3)
      if (err) throw err;
  todoDb.get(
        {'skip': 0, 'limit': 25},
        function(err, todos) {                                       (4)
          if (err) throw err;
          assert.notEqual(                                           (5)
```

```
            todos.length,
            0,
            'Failure: there should be one todo'
          );
          testsCompleted++;                                    ❻
          cb();
        }
      );
    });
  });
}
```

❶ Note that test has started
❷ Delete any existing items
❸ Add item
❹ Get items
❺ Assert that items exist
❻ 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. The above example shows the use of a `notEqual` assertion. All todo items are deleted, an item is then added, and the application logic then gets all items. If the number of items is 0, the assert will fail and an exception will be thrown.

In addition to `equal` and `notEqual` functionality, the assert module offers strict versions of the above called `strictEqual` and `strictNotEqual`. These use the strict equality operator ("===") rather than the more permissive "==". For the comparison of 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.

Next add a test of the todo application's check to see if any tests remain unfinished.

Listing 6.4 test.js Test to see if check for unfinished todos works.

```
function todosUnfinishedCheckTest(cb) {
  testsStarted++;                                              ❶
  todoDb.delete(function() {                                   ❷
    todoDb.add(                                                ❸
      {'item': 'feed bobcat'},
      function(err, todo) {
```

```
        if (err) throw err;
        todoDb.unfinished(function(unfinished) {          4
          assert.ok(                                      5
            unfinished,
            'Failure: todos should be unfinished'
          );
          testsCompleted++;
          cb();
        }
      );
    });
  });
}
```

1. Note that test has started
2. Delete any existing items
3. Add item
4. Check for unfinished items
5. Assert that unfinished items exist

The `ok` assertion provides an easy way to test a value for being `true` as shown by the previous example.

Now that you've defined tests, you can add logic to the file to actually run the tests.

Listing 6.5 test.js Running the tests and reporting test completion.

```
deleteTest(function() {
  addTest(function() {
    todosUnfinishedCheckTest(function() {
      console.log(                                        1 Indicate completion
        'Completed '
          + testsCompleted
          + ' of ' + testsStarted
          + ' tests.'
      );
      todoDb.close();                                     2 Close database connection
    });
  });
});
```

The previous logic will run each test, print how many tests were ran and completed, then will close the database connection. 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 started and completed. Keeping track of started and completed tests is a safty precaution against flaws in individual tests where the test may execute without reaching the assertion.

In addition to the above functionality, the assert module can also be used to check that thrown error messages are correct, as the following example shows. The second argument in the `throws` call is a regular expression that looks for the text "Database" in the error message.

```
assert.throws(
  function() {
    throw new Error("Database error");
  },
  /Database/
);
```

Now that you've learned how to write automated tests for your applications using only Node's built-in functionality, let's now look at how you can make things easier using a third-party unit testing framework.

6.1.2 Expresso

There are a number of excellent testing frameworks that have been created by the Node community: shoulda.js, qunit, spec, nodeunit, Expresso, and more. 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're going to first look at Expresso[2] as it's easy to learn and ships with HTTP-specific assertion utilities. Expresso also extend's Node's "assert" module, adding a number of additional assertion tests. In addition to basic unit testing, Expresso is also capable of issuing requests against an HTTP server and performing assertions against the response header and body data.

Footnote 2 https://github.com/visionmedia/expresso

Enter the following command to install Expresso.

```
$ npm install -g expresso
```

Once installed, you'll have a new command-line tool: `expresso`. This type of command is referred to in the testing world as a "test runner" and makes it easier to manage the running of tests. This command, when executed inside a project directory, will run a project's tests.

To add Expresso tests to your project, simply create a directory named "test"

and add one or more files in which you specify your tests. You can name these files whatever you'd like, provided they end with ".js".

Let's say you want to port the tests we wrote earlier for the todo application to Expresso. You'd change to the application's directory then create a `test` directory using the following command.

```
mkdir test
```

Then, in this directory, you would make an Expresso test file called `expresso_test.js` containing the testing logic in listing 6.6. For brevity's sake we've only ported over one of the tests.

Listing 6.6 expresso_test.js Using Expresso to test application functionality

```
var assert = require('assert')
  , todo = require('../lib/todo')
  , todoDb = new todo.TodoDb();

module.exports = {                                                    (1) Export tests
  'delete test': function() {                                         (2) Name of test
    todoDb.delete(function() {
      todoDb.get({'skip': 0, 'limit': 25}, function(err, todos) {
        if (err) throw err;
        assert.equal(                                                 (3) Test assertion
          todos.length,
          0,
          'Failure: there should be no todos after deleting them'
        );
        todoDb.close();
      });
    });
  }
}
```

In the test file you populate `module.exports` with one or more tests. Each test is named and assigned logic. The Node assert functionality is no different.

If you wanted to complete the database-driven version of the todo application by adding a web interface, Expresso would allow you to test it. To illustrate this, create a file named `todo_server.js` in the `lib` directory and add the following logic. This file is a Node module defining HTTP server functionality that works with the todo application logic defined earlier in the chapter.

Listing 6.7 todo_server.js Web interface to todo application

```
var todo = require('./todo')
  , todoDb = new todo.TodoDb()
  , http = require('http')
  , qs = require('querystring');

exports.server = http.createServer(
  function(req, res){
    if ('/' == req.url){
      switch (req.method){
        case 'GET':
          exports.show(res);
          break;
        case 'POST':
          exports.create(req, res);
          break;
      }
    }
  }
);

exports.show = function(res) {
  var todos = todoDb.get(
    {'skip': 0, 'limit': 25},
    function(err, todos){
      var html = '<h1>Todo list</h1>'
        + '<ul>';

      var html = html + todos.map(function(todo){
        return '<li>' + todo.item + '</li>';
      }).join("\n");

      html = html + '</ul><p />'
        + '<form method="post" action="/">'
        + '<p><input type="text" name="item" /></p>'
        + '<p><input type="submit" value="Add Item" /></p>'

      res.setHeader('Content-Length', html.length);
      res.setHeader('Content-Type', 'text/html');
      res.end(html);
    }
  );
}

exports.create = function(req, res){
  var data = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk){
    data += chunk;
  });
  req.on('end', function(){
    var todo = qs.parse(data);
    todoDb.add(todo, function(err, todo) {
      exports.show(res);
    });
```

❶ **Route requests**

❷ **Show todos**

❸ **Create todo**

```
  });
}
```

Next, create a file `server.js` containing the following code.

```
var todo = require('./lib/todo_server.js')
  , http = require('http')
  , port = 3000;

todo.server.listen(port);

console.log('Listening on port ' + port);
```

You can now start a todo web application server by entering the following command.

```
node server.js
```

Now quit the server. Expresso can test HTTP server objects directly so there's no need to actually have one running.

Following is an another example test file that demonstrates Expresso's ability to test HTTP server functionality using Expresso's custom `response` assertion.

Listing 6.8 todo_server_test.js Using Express to test web server functionality

```
var assert = require('assert')
  , http = require('http')
  , server = require('../lib/todo_server').server;

module.exports = {

  'server running': function(beforeExit) {
      assert.response(
        server,
        {url: '/'},
        function(res) {
          assert.notEqual(
            res.body.indexOf('<h1>Todo list</h1>'),
            -1,
            'Failure: server not responding');
        }
      );
  }
}
```

1. **HTTP server**
2. **Request object**
3. **Reponse function**

Expresso's `assert.response` takes as arguments an HTTP server, a request

object, and response object or function, and an optional callback to be called upon completion of the assertion.

By default Expresso will execute all tests in parallel, making test suites run extremely quickly. Writing tests in parallel requires you to, however, make sure what one test does doesn't interfere with another test's execution. If you're willing to sacrifice testing speed for simplicity you can alter Expresso's default behaviour so tests run sequentially by passing the `--serial` flag, as the following command example shows. Whether you run tests in parallel or serially is a matter of personal preference.

```
expresso --serial
```

For more about Expresso, check out its full online documentation[3].

Footnote 3 http://visionmedia.github.com/expresso/

Next we'll be looking at Nodeunit which, unlike Expresso, executes all tests serially.

6.1.3 Nodeunit

Nodeunit[4] is a Node unit testing framework that can also be used to test client-side code in the browser. As mentioned previously Nodeunit executes tests sequentially, and while this can be much slower than executing tests in parallel, it can also lead to simpler tests, and easier debugging.

Footnote 4 https://github.com/caolan/nodeunit

Enter the following to install nodeunit:

```
npm install -g nodeunit
```

Once complete you'll have a new command available named `nodeunit`, much like `expresso`. 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.

To add nodeunit tests to your project, simply create a directory for them (the directory is usually named "test"). As with Expresso, each test script should populate `module.exports` with tests.

Following 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 above 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.

Once each function exported by the test script has completed its assertions, 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 guards against tests that pass when they shouldn't because assertions haven't been fired.

Why wouldn't assertions fire? When writing unit tests, there is always the danger 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 wanted to safeguard against this, you could manually implement an assertion counter such as the one shown in listing 6.9.

Listing 6.9 manual_assertion_counter.js Manually counting assertions

```
exports.testPony = function(test) {
  var assertionCounter = 0;
  if (false) {
    test.ok(false, 'This should not have passed.');
    assertionCounter++;
  }
  test.ok(true, 'This should have passed.');
  assertionCounter++;
  test.equal(
    assertionCounter,
    2,
    'Not all assertions triggered.'
  );
  test.done();
```

❶ **Count assertions**

❷ **Increment assertion count**

❸ **Increment assertion count**

❹ **Test assertion count**

```
}
```

This gets tedious, however. 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 unneccesary 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();
}
```

Nodeunit, as we mentioned earlier, also allows you to test client-side Javascript. Before using this functionality, however, nodeunit requires you to run a script to assemble the client-side version using components of the server-side version. To see this in action you first need to clone the nodeunit Git repository then generate the browser-compatible Javascript file using the following commands.

```
git clone https://github.com/caolan/nodeunit.git
cd nodeunit
make browser
```

The `make browser` command generates the browser-compatible version of nodeunit and places it in the `dist/browser` folder. The CSS needed by the browser tester is available in the `share folder`.

Following is some example HTML showing how nodeunit can be incorprated into HTML. To try this out, put this HTML into a file called `test.html` in the directory created when you cloned the nodeunit repo.

Listing 6.10 test.html

```
<html>
  <head>
    <title>My web application</title>
    <link rel="stylesheet" href="share/nodeunit.css" type="text/css" />
    <script src="dist/browser/nodeunit.js"></script>          (1) Load nodeunit
    <script src="js/test/tests.js"></script>                  (2) Load tests
  </head>
  <body>
    <script>
      nodeunit.run({                                          (3) Run tests
```

```
        'Pony tests': tests,
      });
    </script>
  </body>
</html>
```

Next, create a directory in which you'll put the test script.

```
mkdir -p js/test
```

Inside this directory, in a file called `tests.js`, put the following test code.

```
this.tests = {
  'testPony': function (test) {
    var isPony = true;
    test.ok(isPony, 'This is not a pony.');
    test.done();
  }
};
```

You should now be able to navigate to the `test.html` file, using a web browser, and see the client-side version of nodeunit in action as illustrated in figure 6.2.

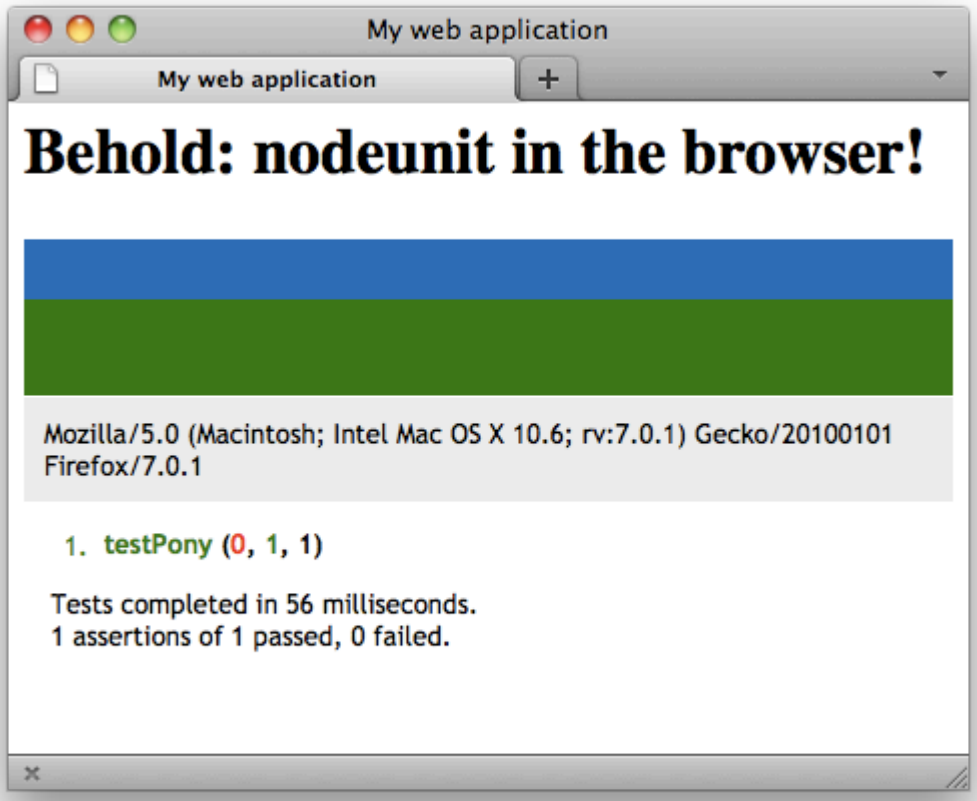

Figure 6.2 The potential of unit testing in the browser with the same tool used for the server-side is definitely appealing.

Whether you want to use TDD-flavoured or BDD-flavoured unit testing is mostly a matter of preference. If non-developer stakeholders are going to be included or you simply prefer a more English-like way of defining tests, you may want to look at BDD-flavoured frameworks.

Now that you've learned how to use TDD-flavored unit testing frameworks, let's look at how you can incorprate a different style of unit testing designed to make your tests easier for to understand, even for non-programmers.

6.1.4 Vows

Vows is Node's most popular BDD-flavoured unit testing framework. Vows tests are more structured than many frameworks, with the structure intended to make the tests easy to read and maintain.

Vows uses its own 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", conceptual areas of concern, to be tested. A context may contain a number of things: a "topic", one or more "vows", and/or one or more related

contexts. A topic is a logic related to a context. A vow is a test of the result of a topic. Figure 6.3 visually represents how Vows structures tests.

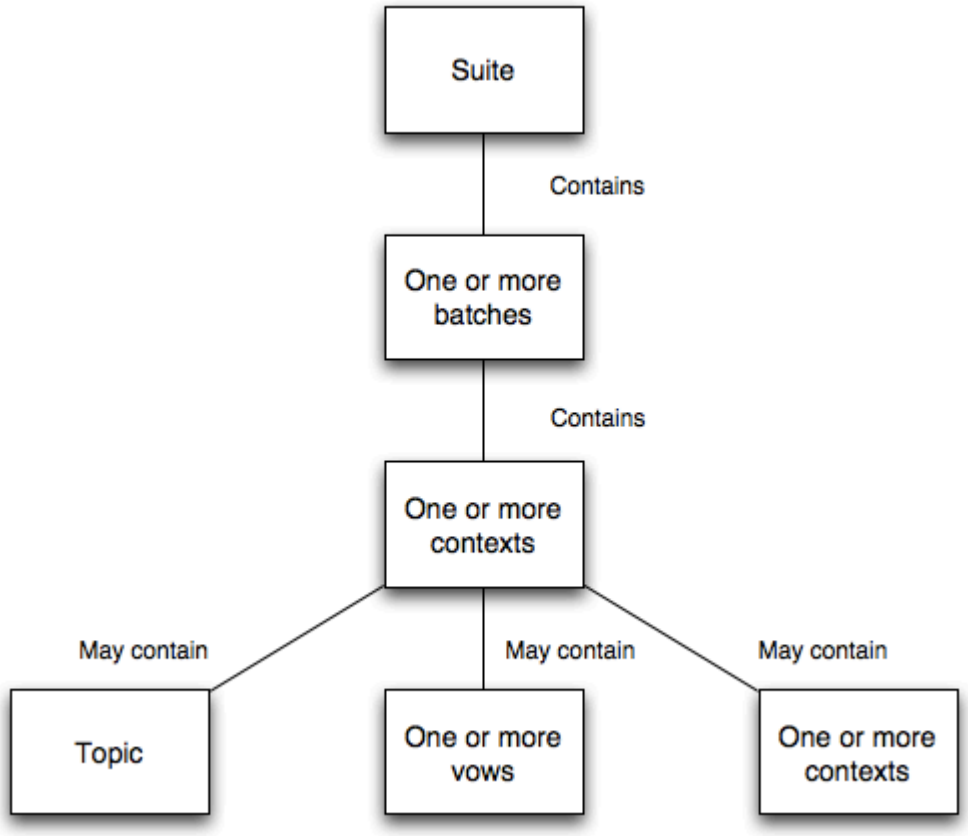

Figure 6.3 Vows structures tests in a suite using batches, contexts, topics, and vows.

Enter the following to install Vows:

```
$ npm install -g vows
```

Testing in Vows can either be triggered by running a script containing test logic or by using the `vows` command-line test runner. The following is an example, using one of the tests of the todo application's core logic, of a stand-alone test script that can be run like any other Node script.

Listing 6.11 vows_test.js Using Vows to test the todo application

```
var vows = require('vows')
  , assert = require('assert')
  , todo = require('./lib/todo')
  , todoDb = new todo.TodoDb();
```

```
vows.describe('todo application').addBatch({                    (1) A batch
  'todo deletion': {                                            (2) A "context"
    topic: function () {                                        (3) A topic
      var callback = this.callback;
      todoDb.delete(function() {
        todoDb.get(
          {'skip': 0, 'limit': 25},
          callback
        );
      });
    },
    'get after deletion': function(err, todos) {                (4) A vow
      assert.equal(todos.length, 0);
      todoDb.close();
    }
  }
}).run();
```

In the above example a batch is created. Within the batch, a context is defined. Within the context, a "topic" and "vow" are defined. Note the use of the callback to deal with asynchronous logic in the topic. If a topic isn't asynchronous a value can just be returned rather than sent via a callback.

If you wanted to include listing 6.11 script in a folder of tests so that it could be run with the Vows test runner, you'd simply change the last line to the following.

```
}).export(module);
```

To run all tests in a folder named "test", you'd enter the following.

```
$ vows test/*
```

For more about Vows, check out the project's online documentation[5], as shown in figure 6.4.

Footnote 5 http://vowsjs.org/

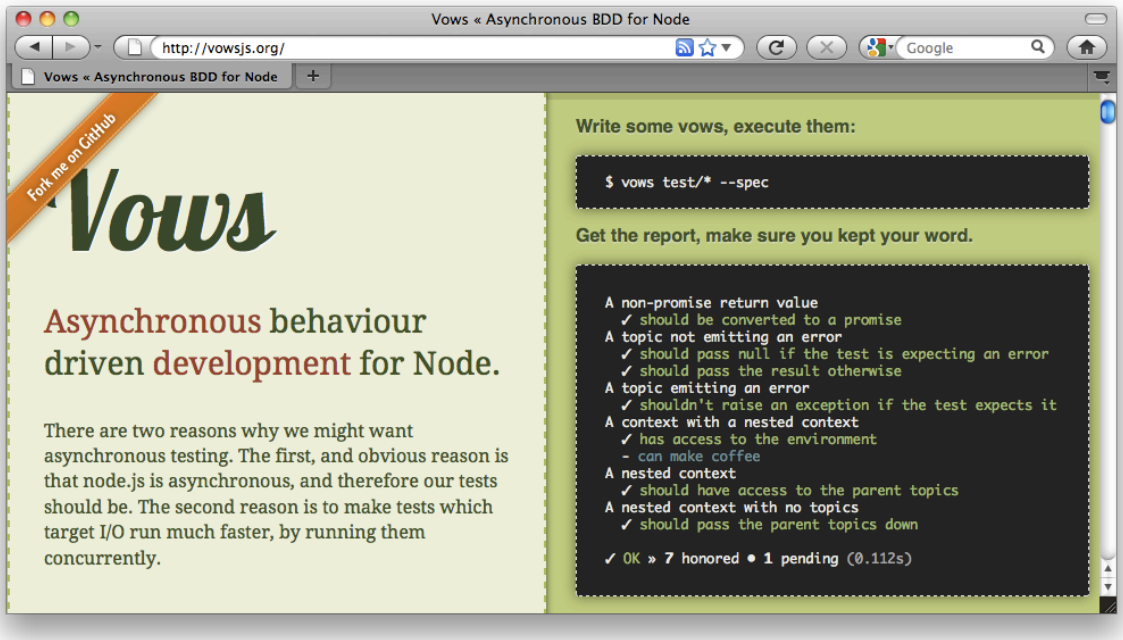

Figure 6.4 Vows combines full-featured BDD testing capabilties with features such as macros and flow control.

While Vows offers a comprehensive testing solution, you might not like the test structure Vows imposes - requiring the use of "batches", "contexts", "topics" and "vows" - or you might like the features of a competing testing framework or be familiar with another testing framework and see no need to learn another. If this is the case, should.js might be a solution.

6.1.5 Should.js

Should.js is a BDD-flavoured assertion library that makes your assertions more readable. It's designed to be used in conjunction with other testing frameworks so you can continue using your chosen framework.

To install should add it as a development dependency to your package.json file:

```
...
"devDependencies": {
  "should": "0.3.x"
}
...
```

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")`.

Let's say you're writing a Node command-line tip calculator that you want to

use to figure out who should pay what when you split the bill with friends. You'd like to write tests for your calculation logic in a way that's more easily understandable by your non-programmer friends, so they don't think you're cheating them.

To set up your tip calculator application, first enter the following commands to set up a folder for it and install Should.js for testing.

```
mkdir -p ~/tmp/tips
cd ~/tmp/tips
mkdir node_modules
npm install should
```

Then create a file named `tips.js` in the `node_modules` folder. This file should contain the logic detailed in listing 6.7, defining the application's core functionality as a module.

Listing 6.12 tips.js Logic for calculating tips when splitting a bill

```
exports.addPercentageToEach = function(prices, percentage) {
  return prices.map(function(total) {
      total = parseFloat(total);
      return total + (total * percentage);
  });
}

exports.sum = function(prices) {
  return prices.reduce(
    function(currentSum, currentValue) {
      return parseFloat(currentSum) + parseFloat(currentValue);
    }
  )
}

exports.percentFormat = function(percentage) {
  return parseFloat(percentage) * 100 + '%';
}

exports.dollarFormat = function(number) {
  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**

The tip calculator logic module includes four helper functions. `addPercentageToEach` increases each number in array by a given percentage.

`sum` calculates the sum of each element in an array. `percentFormat` formats a percentage for display. Finally, `dollarFormat` Formats a dollar value for display.

Now, before creating a command-line interface to the logic, create the following test script for it. This script, containing BDD-style assertions, is still a bit tricky for non-programmers to read but still easier for you to guide your friends through than conventional assertion logic.

Listing 6.13 tips_test.js Logic so calculating tips when splitting a bill

```
var tips = require('tips')
  , should = require('should')
  , tax = 0.12
  , tip = 0.15;

var prices = [10, 20];
var withTipAndTax = tips.addPercentageToEach(prices, tip + tax);
withTipAndTax.should.eql([12.7, 25.4]);
var rounded = tips.sum(withTipAndTax).toFixed(2);
rounded.should.equal('38.10');
tips.dollarFormat(rounded).should.equal('$38.10');
tips.percentFormat(0.15).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**

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. 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 tips_test.js
```

Should.js supports many types of assertions -- everything from assertions using regular expressions to assertions that check object properties -- allowing comprehensive testing of data and objects generated by your application. The project's Github page[6] provides comprehensive documentation of Should.js' functionality.

Footnote 6 http://github.com/visionmedia/should.js

Having looked at unit testing, lets 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. So once you've created a suite of unit tests for your project, acceptance testing provides an additional level of protection against bugs that unit testing might not have covered.

Acceptance testing is similar, conceptually, to testing by end users following a list of things to test. Being automated, however, it's 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 abilities to browse content to selected 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 is logged 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. While Soda provides the benefit of hardnessing real browsers for acceptance testing, Tobi, which we'll look at next, is easier to learn and get up and running on.

6.2.1 Tobi

Tobi[7] is an easy-to-use acceptance testing framework that emulates the browser and leverages should.js, offering access to its assertion capabilities. The framework leverages two third-party modules, `jsdom` and `htmlparser`, to simulate a web browser, allowing access to a virtual DOM.

Footnote 7 https://github.com/LearnBoost/tobi

In the world of client-side Javascript development, web developers often use the JQuery[8] library when they need to manipulate the DOM. JQuery can also be used on the server-side, and Tobi's use of JQuery minimizes the learning required to create tests with it. Tobi can test external sites over the network or can interface directly with Express/Connect applications.

Footnote 8 http://jquery.com/

Enter the following to install Tobi:

```
$ npm install tobi
```

The following test script 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.

Listing 6.14 tobi_remote_test.js Testing a remote site's login capability using Tobi

```
var tobi = require('tobi')
  , browser = tobi.createBrowser(3000, '127.0.0.1');

browser.get('/', function(res, $){
  $('form')
    .fill({ item: 'Floss the cat' })
    .submit(function(res, $) {
      $('ul')
      .html()
      .indexOf('Floss the cat')
      .should.not.equal(-1);
    });
});
```

1 **Create browser**
2 **Get todo form**
3 **Fill in form**
4 **Submit data**

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.

If you want to test an Express application you've built, rather than a remote site, it's similarly straightforward. Following 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 6.15 app.js Example Express web application

```
var express = require('express')
  , app = express.createServer();

app.get('/about', function(req, res) {
  res.send(
```

1 **Send HTML**

```
    + '<html>'                                                    response
    +   '<body>'
    +     '<div>'
    +       '<h1>About</h1>'
    +     '</div>'
    +   '</body>'
    + '</html>'
  );
});

module.exports = app;
```

You can test the above application without even running it. The following Tobi test shows how you'd do this.

```
var tobi = require('tobi')
  , app = require('./app')
  , browser = tobi.createBrowser(app);

browser.get('/about', function(res, $){
  res.should.have.status(200);
  $('div').should.have.one('h1', 'About');
  app.close();
});
```

Tobi includes no test runner, but can be used with unit testing frameworks such as Expresso or Nodeunit.

6.2.2 Soda

Soda[9] takes a different approach to acceptance testing. While other Node acceptance testing frameworks simulate browsers, Soda remote controls real browsers. Soda, as shown in figure 6.5, does this by sending instructions to Selenium Server (also known as Selenium RC) or the Sauce Labs "Sauce Cloud" on-demand testing service.

Footnote 9 https://github.com/LearnBoost/soda

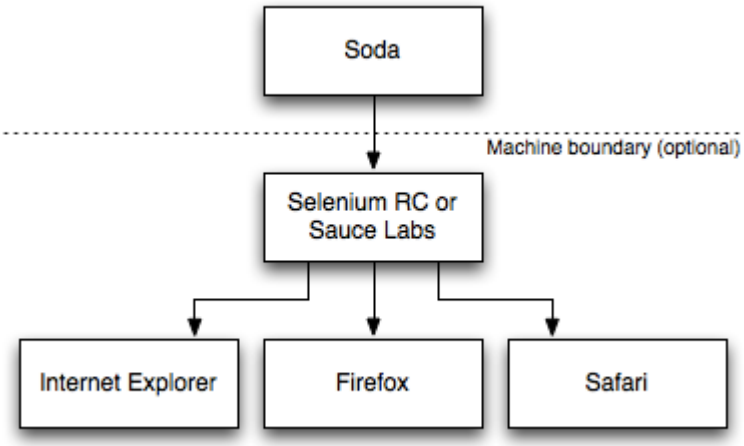

Figure 6.5 Soda is an acceptance testing framework that allows real browsers to be remote controlled. Whether using Selenium RC or the Sauce Labs service, Soda provides an API that allows a Node to direct testing that take into account the realities of different browser implementations.

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 actual 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 then using Sauce Cloud may be desirable.

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
```

If your machine has Java installed, Selenium Server is painless to install. All you have to do is download a recent ".jar" file from the Selenium Downloads page [10]. Once you've downloaded the file you can run the server using a command similar to the following.

Footnote 10 http://seleniumhq.org/download/

```
java -jar selenium-server-standalone-2.6.0.jar
```

Once the server is 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 should be opened in the browser for testing and the `browser` specifies the browser to be used for testing.

```
var soda = require('soda')
  , assert = require('assert');

var browser = soda.createClient({
    host: 'localhost'
  , 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('undefined: undefined', cmd, args.join(', '));
});
```

Next in your test script should be the tests themselves. Following 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. Operations like "clickAndWait" are referred to by the Selenium community as "Selenese" and are documented online[11].

Footnote 11 http://release.seleniumhq.org/selenium-core/1.0.1/reference.html

Listing 6.16 A Soda test allows you to enter "Selenese" 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){
    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

If you go the Sauce Cloud route, simply sign up for the service as the Sauce Labs[12] website and change the code in your test script that returns `browser` to something like the following.

Footnote 12 https://saucelabs.com/

Listing 6.17 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. Desired browser type
5. Desired browser version
6. Make test fail if it takes too long

And that's all. 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 back into your codebase and can develop with greater confidence.

For those new to unit testing, Expresso and Nodeunit are excellent frameworks to start with: they're easy to learn, flexible, and can work with should.js if

BDD-style tests are desired. For those who like the BDD approach and seek a system of structuring tests and controlling flow Vows 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 get started in and developers familiar with JQuery will be up and running easily. For those requiring acceptance testing that takes into account browser discrepancies, Soda may be worth running, but testing is slower and you must learn "Selenese".

Now that you've got a handle on how automated testing can be conducted in Node, you'll next learn about Connect, an HTTP middleware framework that helps you avoid reinventing the wheel in your web applications.

7 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. Connect's main components are the "dispatcher" that makes use of the middleware, and several bundled middleware that you can use in your applications such as request logging, static file serving, and session managing. Connect serves as an abstraction layer, preventing the need to constantly re-invent the wheel for developers who wish to create their own frameworks. Connect is flexible enough that building any type of web application is possible, whether it's your personal blog or the next big social networking website, Connect has you covered.

Connect is bundled with a variety of middleware for common needs, but it's also very easy to define your own! Connect uses middleware because they are designed to be modular and reusable across the applications you design, and we will be looking at how you can create middleware for routing requests, rewriting urls, and handing user and application errors. In figure 7.1 you can see how a Connect application is composed of the dispatcher, as well as any arrangement of middleware.

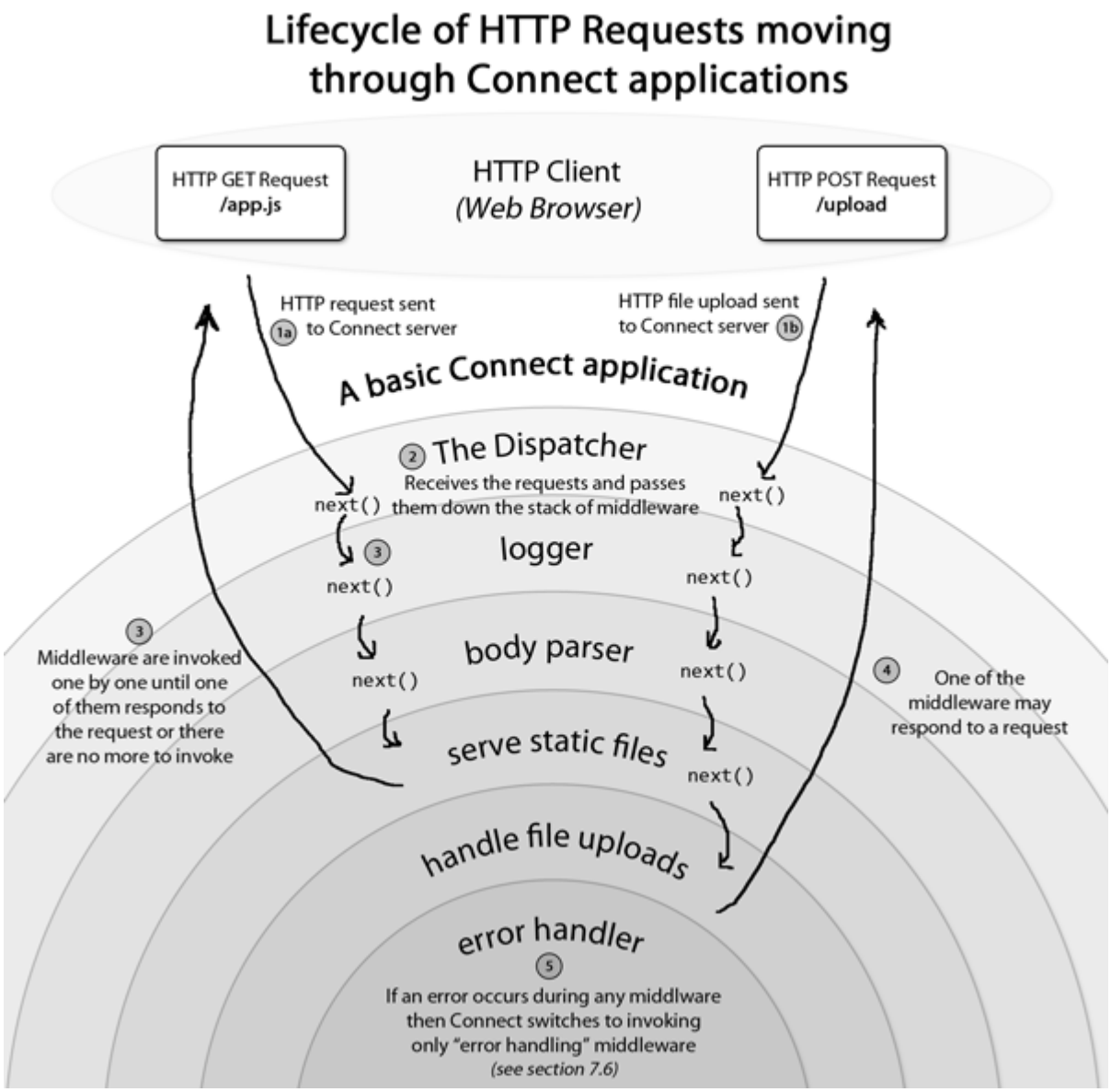

Figure 7.1 The lifecycle of two HTTP requests making their way through a Connect server

The concepts and middleware discussed in this chapter are directly applicable to the higher-level framework "Express" as it uses the same middleware concept. 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). Even though you will likely be using Express to

build your application, you will still benefit from learning about how middleware work because Express embraces the same middleware philosohpies, and Connect's built-in middleware are expected to be used in conjuction with Express.

In this chapter we will walk you through the key things it takes to build a Connect application:

- The initial setup
- How connect middleware work
- Middleware ordering, and why it matters
- How to mount middleware and servers
- How to create configurable middleware
- Error handling middleware

First up, let's create a basic Connect application.

7.1 Setting up a Connect application

Connect is a third party module, so 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 you start off by requiring the `connect` module which returns 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. Well 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" code by the dispatcher.

Listing 7.1 minimal_connect.js: A minimal Connect application

```
var http = require('http')
  , connect = require('connect');
```

```
<!-- -->
var app = connect();
http.createServer(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 "Not Found" indicating that this application is not configured to handle the requested url. This is the first example of how Connect's distpatcher 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 a "middleware" is a simply a JavaScript `function` that by convention accepts three arguments: a request object, a response object, and a new third argument commonly named "next", which is a callback function. This concept 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 A 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 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 back to the

next middleware. So for this `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 pass control back to the distacher which would 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 aptly named `.use()` method, passing it the middleware function:

```
var http = require('http');
var connect = require('connect');
var app = connect();
app.use(logger);
http.createServer(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" 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 actually 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 http = require('http');
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');
}
<!-- -->
var app = connect()
  .use(logger)
  .use(hello);
```

1. **prints out the HTTP method and request URL**
2. **responds 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 the `app` 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);
```

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 makes few assumptions 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 by other web frameworks.

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:

- A middleware can stop the execution of remaining middleware when next() is not called
- You can use the powerful middleware-ordering feature to your advantage
- You can leverage middleware to perform authentication

7.3.1 When middleware don'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 http = require('http');
var connect = require('connect');
<!-- -->
function logger(req, res, next) {                    ❶
  console.log('%s %s', req.method, req.url);
  next();
}
<!-- -->
function hello(req, res) {                           ❷
  res.setHeader('Content-Type', 'text/plain');
  res.end('hello world');
}
<!-- -->
var app = connect()
  .use(hello)                                        ❸
  .use(logger);
```

❶ always calls next() so subsequent middleware are invoked
❷ does not call next() since it actually responds to the request
❸ 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.

NOTE **The moral of next() in one sentence**
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.

7.3.2 Using middleware ordering to your advantage

Let's consider another example. Suppose you wanted to log requests, serve static files, and respond to remaining requests with "hello world". You might setup the middleware as shown in the following snippet (note that `serveStaticFiles` is a theoretical middleware that serves static files).

```
var app = connect()
  .use(logger)
  .use(serveStaticFiles)
  .use(hello);
```

Now suppose you only want to log static file requests in production, but don't want to log them in development. You could move `serveStaticFiles` above `logger`, which would respond with the static file when the requested url has a matching file, otherwise it would call `next()` and continue on down through the remaining `logger` and `hello` middleware. Moving `serveStaticFiles` above `logger` effectively prevents the static file requests from being logged, but the file would still be served as expected:

```
var app = connect()
  .use(serveStaticFiles)
  .use(logger)
  .use(hello);
```

That is just one example of how the difference between calling `next()` in a middleware or actually responding to the request in a middleware can be used to

your advantage.

7.3.3 Leveraging middleware to perform 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 should grant access to a file named "/secret.txt" only to logged in users. The `restrictFileAccess` middleware could check if the requested url path is "/secret.txt", and then make sure the provided username and password are valid. Only when both of those criteria are true will the middleware call `next()` continuing on to the static file server, otherwise it responds with a "403 Forbidden" status code. 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');
<!-- -->
var app = connect()
  .use(logger)
  .use(restrictFileAccess)
  .use(serveStaticFiles)
  .use(hello);
```

❶ **next() will only be called if the username and password are 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 helps 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 its `req.url` will be adjusted so the "/blog" prefix is removed. This allows for seamless reusability with existing node HTTP servers or middlware.

Let's illustrate this functionality with an example. It is common for applications to have their own administration area so that sthe 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 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 http = require('http');
var connect = require('connect');
<!-- -->
var app = connect()
  .use(logger)
  .use('/admin', restrict)                                    ❶
  .use('/admin', admin)
  .use(hello);
<!-- -->
http.createServer(app).listen(3000);
```

❶ 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 in section

7.2 with an admin area by using mounting and adding two new middleware:

- A `restrict` middleware that ensures a valid user is accesing 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 auth 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 as shown.

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(' ')
    , scheme = parts[0]
    , auth = new Buffer(parts[1], 'base64').toString().split(':')
    , user = auth[0]
    , pass = auth[1];
<!-- -->
  if ('tobi' == user && 'ferret' == pass) {                    (1)
    next();                                                    (2)
  } else {
    next(new Error('Unauthorized'));
  }
}
```

(1) "tobi:ferret" is the only valid user entry in this case
(2) a real authentication middleware would check a database

Again, notice how this middleware does not do any checking of `req.url` to

ensure that "/admin" is what is actually being requested, because Connect and mounting are handling this for you. As you can see, this allows you to write generic middleware, so this `restrict` middleware could be used to authenticate another part of the site or another application enitrely. When authorization is complete, and the user is in fact "tobi", 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` middlware 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 retreive them from a database.

```
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 "/index.html" and "/aricle/whatever", instead of "/blog/index.html". 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".

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(1)`. You can see that regular urls other than "/admin" will invoke the `hello` middleware as expected:

```
$ curl http://local
hello world
$ curl http://local/foo
hello world
```

You can also see how the `restrict` middleware will return an error to the user when no credentials are given or incorrent credentials are used:

```
$ curl http://local/admin/users
Internal Server Error
$ curl http://jane:ferret@local/admin/users
Internal Server Error
```

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 http://tobi:ferret@local/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 you 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. Reusablity 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) {
  // additional middleware initialization here
  return function(req, res, next) {
    // "options" or anything else defined in the
    // setup "closure" are available inside the middleware
  }
}
```

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 slug names 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 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 finished executing, 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 7.7.

Listing 7.7 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(_, property){      (4)
      return req[property];
    });
    console.log(str);                                             (5)
<!-- -->
    next();                                                       (6)
  }
}
<!-- -->
module.exports = setup;                                           (7)
```

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 compilcated 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 7.8, 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 7.8 connect-router-usage.js: What usage of the 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 of "routes" with URLs and functions the first level in the "routes" object is the request methods to map to
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 acutally 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.

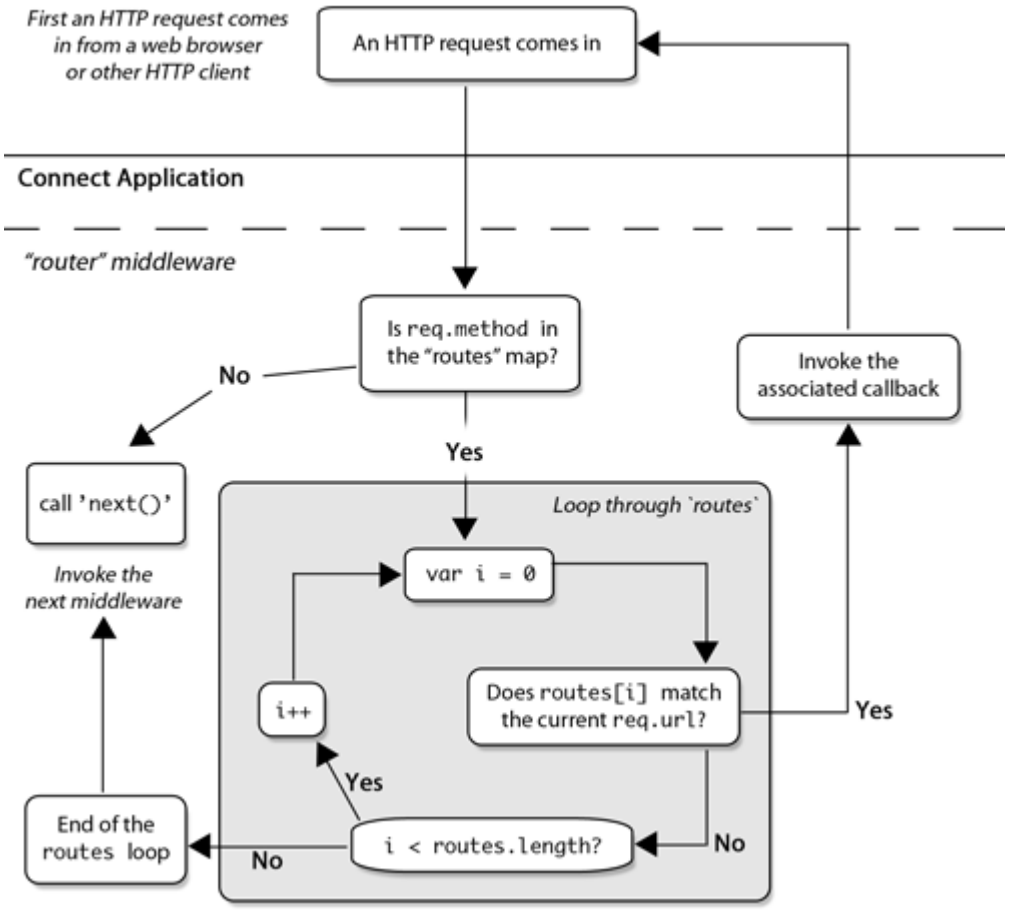

Figure 7.2 Flowchart of the "router" middleware's logic

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 entirey in listing 7.9.

Listing 7.9 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
      return next();
    }
<!-- -->
    var routes = obj[req.method]                               2
      , url = parse(req.url)                                   3
      , paths = Object.keys(routes)                            4
```

```
      , captures
      , path
      , fn;
<!-- -->
    for (var i = 0; i < paths.length; i++) {                    5
      path = paths[i];
      fn = routes[path];
<!-- -->
      path = path
        .replace(/\//g, '\\/')
        .replace(/:(\w+)/g, '([^\\/]+)');
<!-- -->
      var re = new RegExp('^' + path + '$');                    6
<!-- -->
      if (captures = url.pathname.match(re)) {                  7
        var args = [req, res].concat(captures.slice(1));        8
        fn.apply(null, args);
        return;                                                 9
      }
    }
<!-- -->
    next();
  }
};
```

1. first check to make sure the req.method is defined
2. object keyed with the paths
3. parse url for matching against the pathname
4. the paths for this http method as an array
5. loop the paths
6. construct the regular expression
7. attempt match against the pathname
8. pass the capture groups
9. 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 futher 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 as it is so far. For example, you could utilize the power of closures to cache the regular expressions which would otherwise be

compiled per-request. But we won't bother with that now.

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')
  , http = require('http')
  , url = require('url');
<!-- -->
var app = connect()
  .use(rewrite)
  .use(showPost);
<!-- -->
http.createServer(app).listen(3000);
```

The following `rewrite` middleware implementation in listing 7.10 parses the url to access the pathname, then matches with a regular expression. The first capture group (the slug) is passed to a theoretical `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 7.10 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) {
  if (path.match(/^\/blog\/posts\/(.+)/)) {                    (1)
    findPostIdBySlug(RegExp.$1, function(err, id){
      if (err) return next(err);                               (2)
      if (!id) return next();                                  (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() and don't do any more processing
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, 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.

In this section 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.

```
function hello(req, res) {
  foo();
  res.setHeader('Content-Type', 'text/plain');
  res.end('hello world');
}
<!-- -->
var app = connect()
  .use(hello);
<!-- -->
http.createServer(app).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 Handing 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 leveage, such stack traces, file names and line numbers.

NOTE	**Use NODE_ENV to set the application "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 7.11, whereas regular middleware take the form `(req, res, next)` as you have already learned.

Listing 7.11 errorHandler.js: Error handling middleware in Connect

```
function errorHandler() {
  var env = process.env.NODE_ENV || 'development';
  return function(err, req, res, next) {
<!-- -->
    res.statusCode = 500;
    switch (env) {
      case 'development':
        res.setHeader('Content-Type', 'application/json');
        res.end(JSON.stringify(err));
        break;
      default:
        res.end('Server error');
    }
  }
}
<!-- -->
var app = connect()
  .use(hello)
  .use(errorHandler());
```

1 **Error handling middleware define 4 arguments**

2 **This example errorHandler behaves differently depending on the "mode"**

When Connect encounters an error it will switch to invoking only error handling middleware as you can see in figure 7.3.

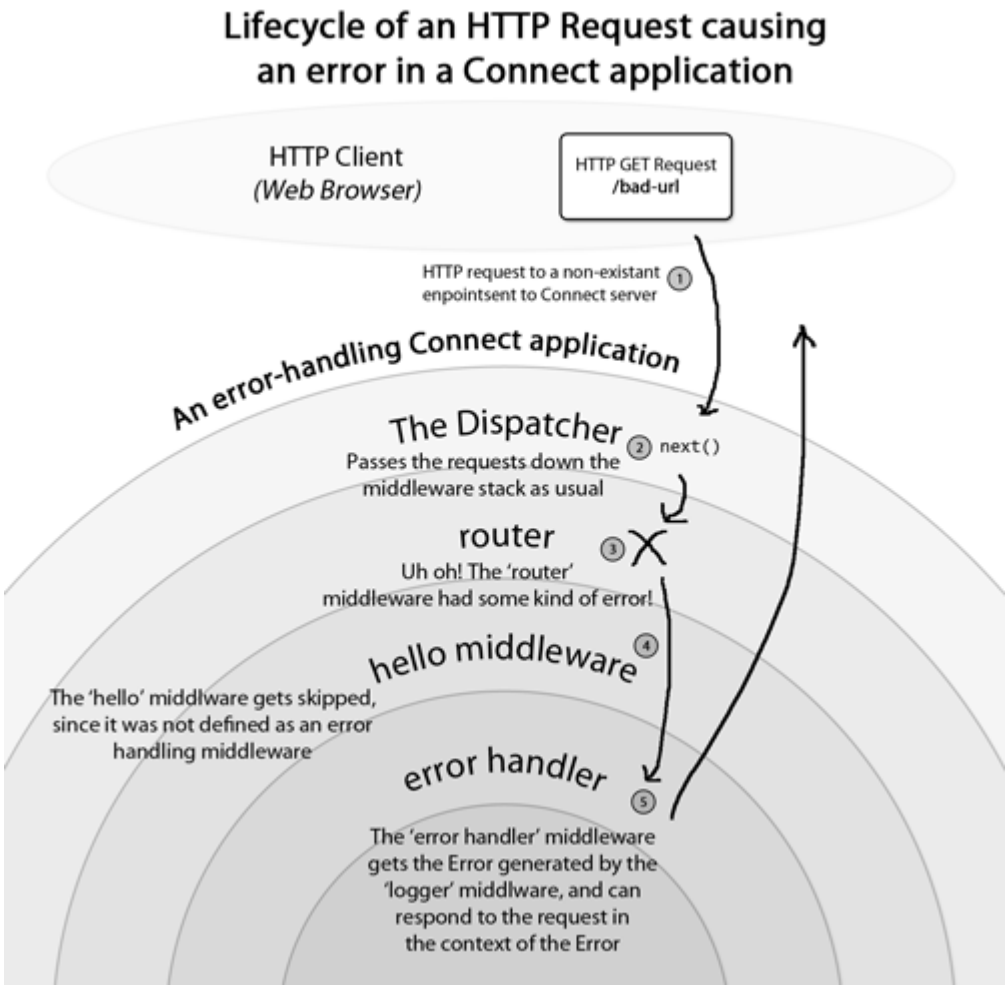

Figure 7.3 The lifecycle of an HTTP request causing an error in a Connect server

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.

```
var app = 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. This application will use a 'users

```
var api = connect()
  .use(users)
  .use(pets)
  .use(errorHandler);
<!-- -->
var app = connect()
  .use(hello)
  .use('/api', api)
  .use(errorPage);
<!-- -->
http.createServer(app).listen(3000);
```

This setup created by the configration for this application is easily visualized in figure 7.4:

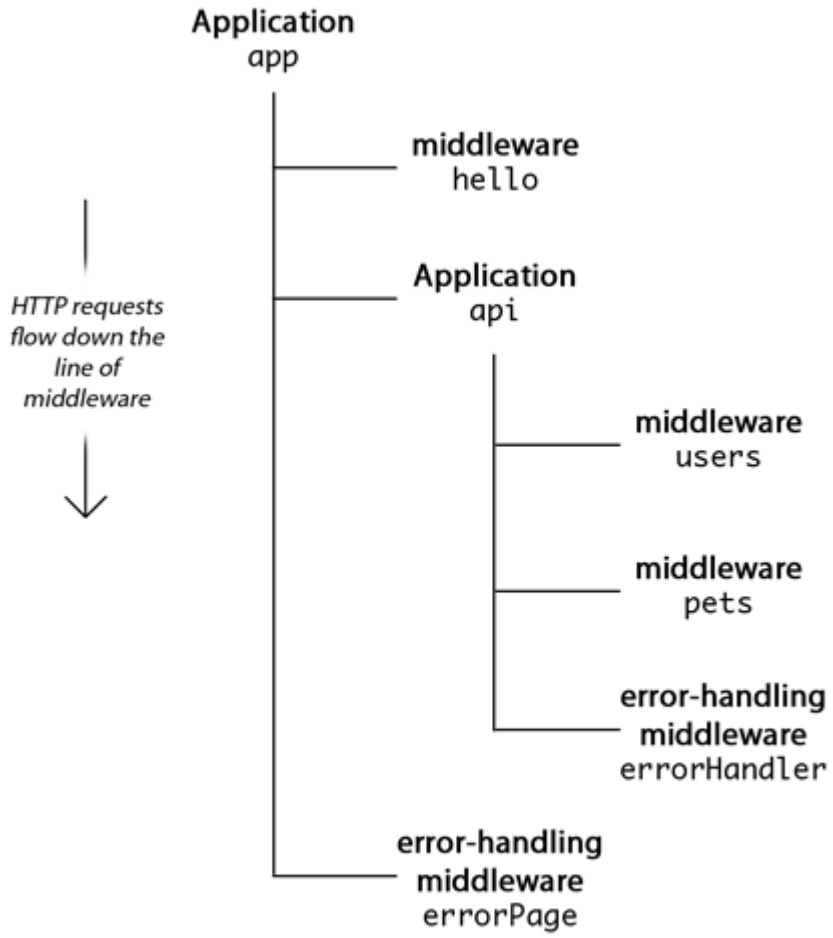

Figure 7.4 Layout of the example application with two error-handling middleware

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 have a reference error, 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 way 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 7.12, we match the `req.url` using a regular expression, then check if the user index exists by using `RegExp.$1` which is the first capture group for the previous 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 7.12 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) {
  if (req.url.match(/^\/user\/(\d+)/)) {
    var user = db.users[RegExp.$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\/(\d+)/)) {
    foo();
  } else {
    next();
  }
}
```

IMPLEMENTING THE "ERRORHANDER" MIDDLEWARE

Finally, you're at the `errorHandler` middleware! Unifying error handling is fantastic for performing content-negotiation based on the `Accept` request header field. However, in this example only a JSON response is supported. We'll be discussing content-negotiation in detail in the Express chapter.

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 resspond with error messages that you know are safe, as the following `errorHandler` implementation does in listing 7.13.

Listing 7.13 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/invaliduser` (where "invaliduser" is not a valid username) 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.

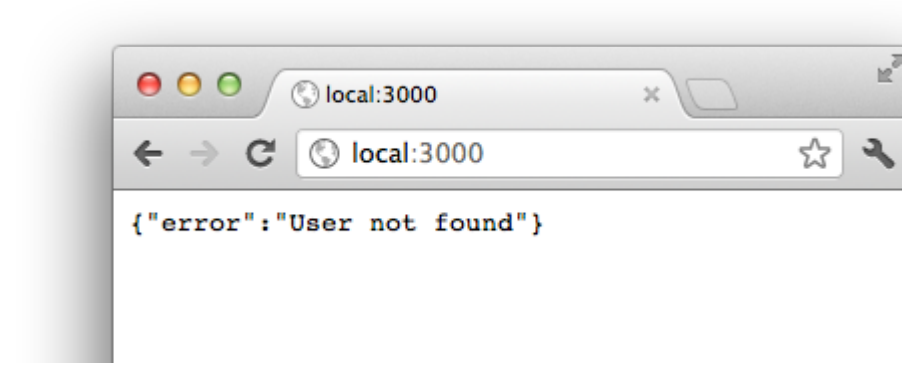

Figure 7.5 The JSON object output of the "User not found" error

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 opportinuty for this middleware to be invoked is by an error occuring in the `hello` middleware. However, that middleware is very unlikely to generate an error, leaving very little chance of this `errorPage` middleware ever being 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(1)` 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. 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.

8 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 varirety 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 most of the more important middleware that you will likely want to use in your own application with an explanation and simple example. Figure 8.1 gives an overview of the middleware that will be covered for quick reference:

Middleware Name	Section #	Summary
cookieParser()	8.1.1	Provides req.cookies and req.signedCookies for subsequent middleware to use.
bodyParser()	8.1.2	Provides req.body for subsequent middleware to use.
limit()	8.1.3	Restricts request body sizes based on a given byte length limit. Must go *before* the bodyParser() middleware.
query()	8.1.4	Provides req.query for subsequent middleware to use.
logger()	8.2.1	Logs configurable information about incoming HTTP requests to a Stream, like stdout or a log file.
favicon()	8.2.2	Responds to the /favicon.ico HTTP requests. Usually placed before logger(), so that you don't even have to see it in your logs.
methodOverride()	8.2.3	Allows you to fake req.method for browsers which are incapable of using the proper method. Depends on bodyParser().
vhost()	8.2.4	Routes to a given middleware and/or Servers based on a specifed hostname (i.e. "example.com").
session()	8.2.5	Sets up an HTTP session for a user and provides a persistent req.session object in between requests. Depends on cookieParser().
basicAuth()	8.3.1	Provides HTTP Basic Authentication to your application.
csrf()	8.3.2	Protects against "Cross-site request forgery" attacks in HTTP forms. Depends on session().
errorHandler()	8.3.3	A middleware useful for development that returns stack traces to the client when a server-side error occurs. *Do not* use for production!
static()	8.4.1	Serves files from a given directory to HTTP clients. Works really well with Connect's "mounting" feature.
compress()	8.4.2	Optimizes HTTP responses using gzip compression.
directory()	8.4.3	Serves directory listings to HTTP clients, providing the optimal result based on the client's Accept request header (plain text, JSON, or HTML).

Figure 8.1 Connect middleware quick reference guide

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 intoo `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 you'll want to pass a secret to `cookieParser()`.

BASIC USAGE

The secret passed in as the first 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 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');
  });
```

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/
{}
{}
```

Now try sending a few cookies. You'll see that both are available as properties of `req.cookies`:

```
$ curl http://localhost/ -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" attacks[1]. 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.DDm3AcVxE9oneYnbmpqxoyhyKsk"`, 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/ -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/ -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/ -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://local/ -H
➥ "Cookie: cart=j:{\"items\":[1]}.sD5p6xFFBO/4ketA1OP43bcjS3Y"
{}
{ cart: { items: [ 1 ] } }
GET / 200 1ms
```

After using the `cookieParser()` middleware in your application, you can write to the `res.cookies` object in order to have Connect set the appropriate "Set-Cookie" headers for you. You can store and 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 small, yet extremely helpful, middleware provides a unified `req.body` property 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 though a JSON request. All you have to do is add the `bodyParser()` middleware before any other middleware that will access the `req.body` object.

```
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://local
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://local
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://local
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 a RAW 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.

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

var app = connect()
  .use(connect.bodyParser());

http.createServer(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"]');
```

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

```
req.end();
```

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) {                                    ❶
  return function(req, res, next){
    var ct = req.headers['content-type'] || '';
    if (0 != ct.indexOf(type)) {                             ❷
      return next();
```

```
    }
    fn(req, res, next);                                          ❸
  }
}

var app = connect()
  .use(type('application/x-www-form-urlencoded', connect.limit('64kb')))
  .use(type('application/json', connect.limit('32kb')))          ❹
  .use(type('image', connect.limit('2mb')))                      ❺
  .use(type('video', connect.limit('300mb')))                    ❻
  .use(connect.bodyParser())
  .use(hello);
```

❶ fn in this case is one of the limit() instances
❷ the returned middleware first checks the content-type
❸ before invoking the passed-in limit() middleware
❹ handes forms, json
❺ image uploads up to 2mb
❻ video uploads up to 300mb

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 accepts no options, and 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 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')
  , http = require('http');

var app = connect()
  .use(connect.logger())                                    (1)
  .use(hello);                                              (2)
```

(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:

- :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 expose the request's content-length. You might define it like this:

```
connect.logger.token('content-length', function(req, res){
  return req.headers['content-length'] || 'unknown';
});
```

The `logger` also comes with other predefined formats than the default one. One of the predefined formats is "dev", which is designed specifically for development where it's often more useful 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' AND 'IMMEDIATE'

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')
  , 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 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);
```

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:

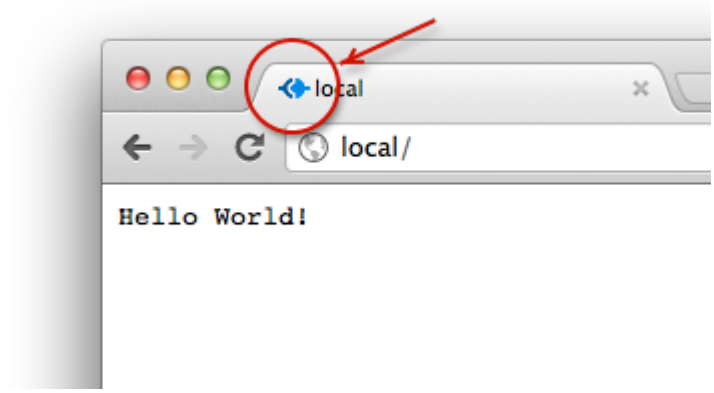

Figure 8.2 Connect's default "favicon"

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');
  });
```

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.:

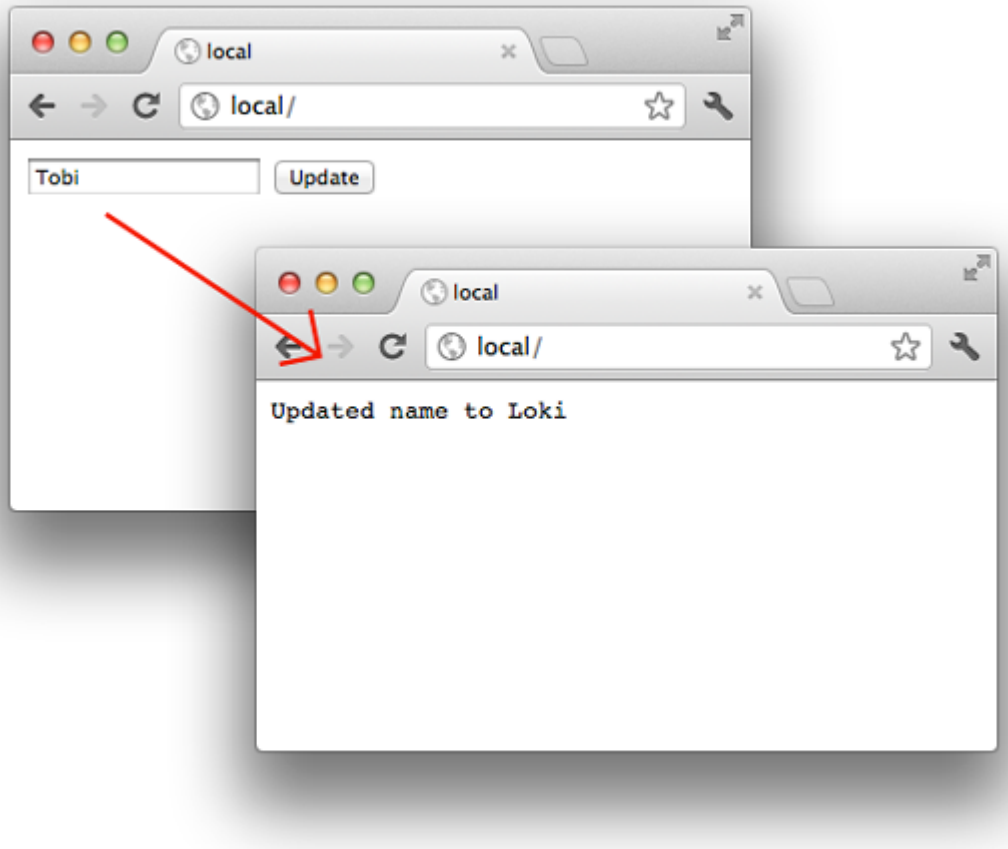

Figure 8.3 An example of using methodOverride to simulate a PUT request updating a form in the browser

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

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);

http.createServer(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

```
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);
```

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 the 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 Server instance to use when the an HTTP request with a matching hostname occurs.

```
var connect = require('connect')
  , http = require('http');

var server = connect()
  , app = require('./sites/expressjs.dev');

server.use(connect.vhost('expressjs.dev', app));

http.createServer(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')
  , http = require('http')
  , fs = require('fs');

var app = connect()
  , sites = fs.readdirSync('source/sites');

sites.forEach(function(site){
  console.log('  ... %s', site);
  app.use(connect.vhost(site, require('./sites/' + site)));
});

http.createServer(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 largest Connect middleware currently 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 the sessions will expire in 4 hours. The cookie is also set to be "httpOnly" meaning client-side scripts cannot access the value. However these are all 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')
  , http = require('http');

var app = connect()
  .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!');
    }
  });

http.createServer(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'
  , { 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 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 get to apply how to apply Redis to store your session data 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')
  , RedisStore = require('connect-redis')(connect)
  , http = require('http');

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 multiple different versions of a module in use 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:

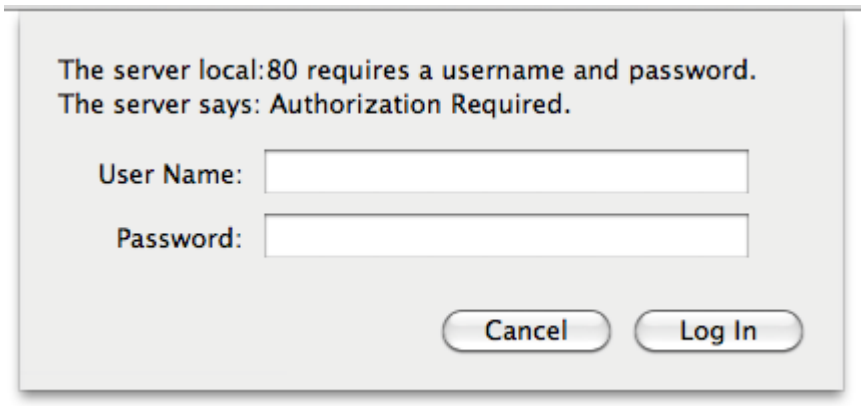

Figure 8.4 Basic authentication prompt

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 }, isValid);     (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')
  , http = require('http');

var app = connect()
  .use(connect.basicAuth('tobi', 'ferret'))
  .use(function (req, res) {
    res.end("I'm a secret\n");
  });

http.createServer(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 http://tobi:ferret@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 user has with the web browser. The attach 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 XSS 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. The default options will be used when no arguments are given:

```
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.

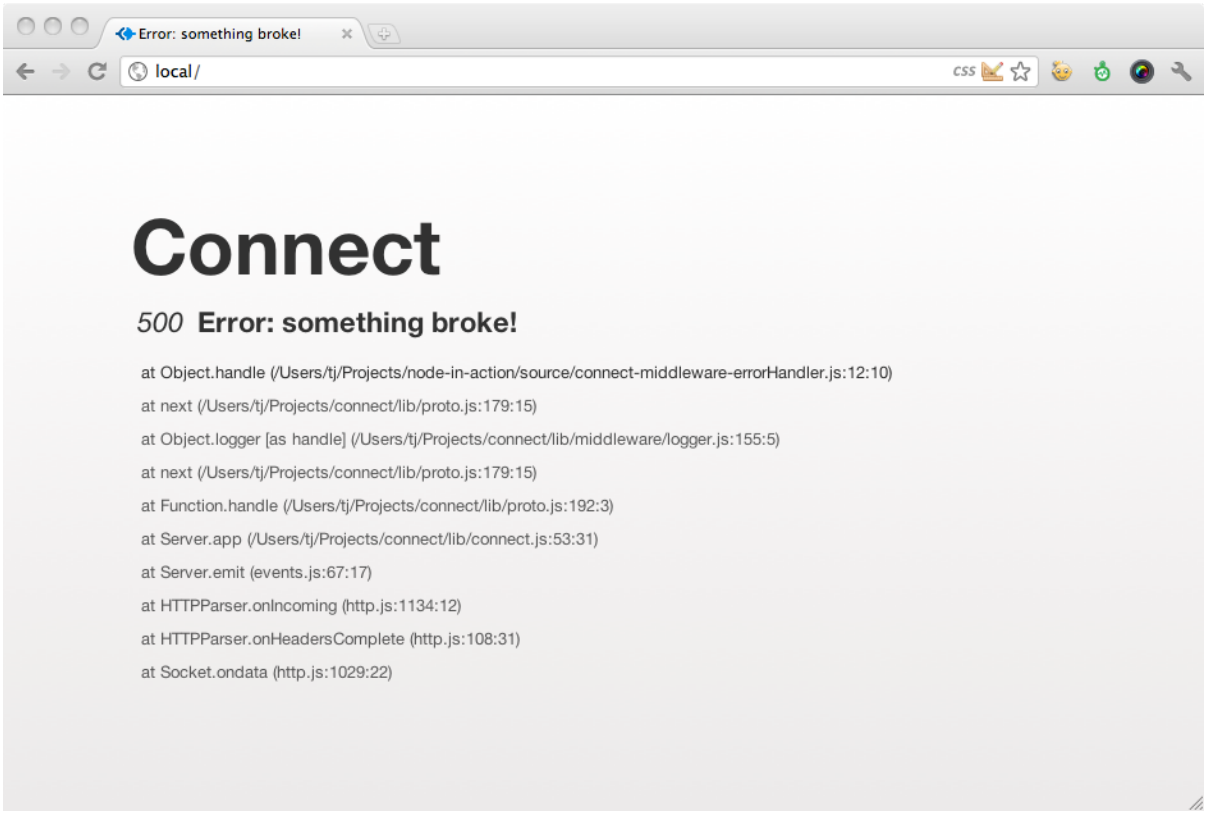

Figure 8.5 Connect's default errorHandler() middleware as displayed in a web browser

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://local/
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're using XMLHttpRequests and Accept JSON you'll get the following response:

```
$ curl http://local/ -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 provides 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 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 bytes[3]. In essense, `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 3 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 pased in the `static()` 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

Node 0.6.0 introduced 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')
  , http = require('http');

var app = connect()
  .use(connect.compress())
  .use(connect.static('source'));

http.createServer(app).listen(3000);
```

In the snippet that follows, a small 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.

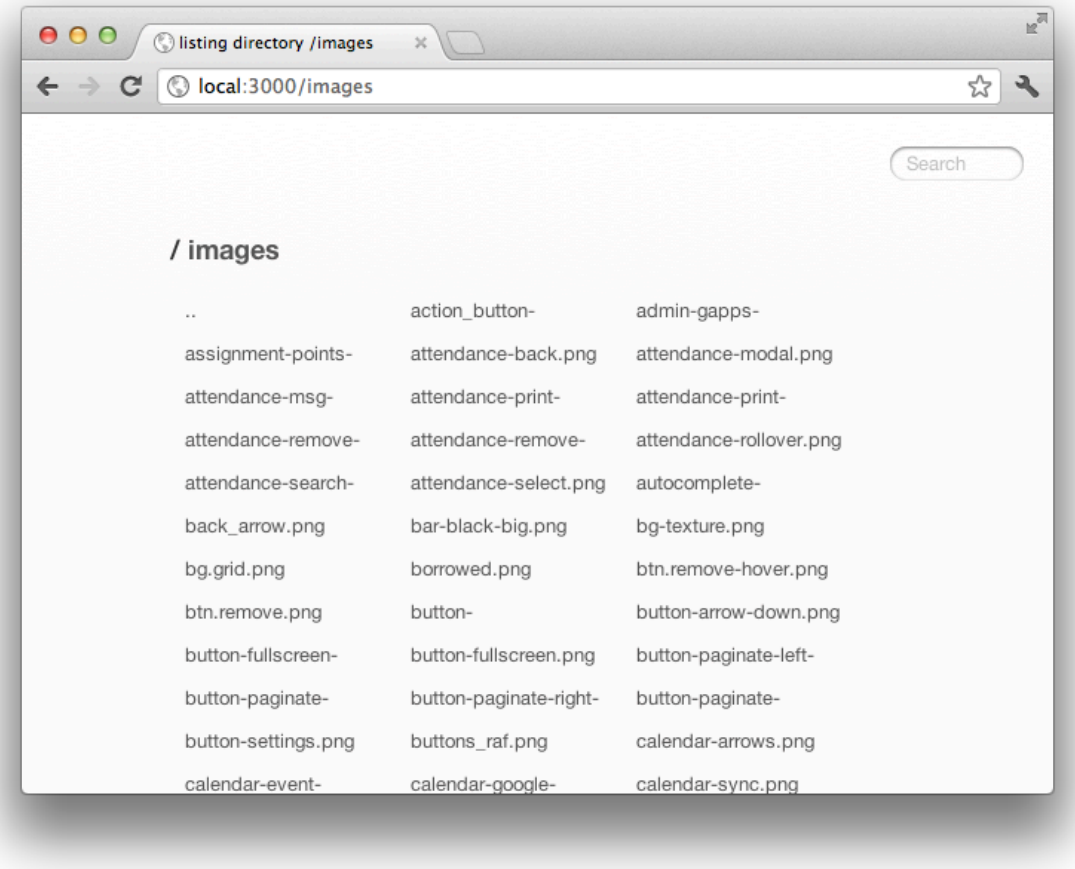

Figure 8.6 Serve directory listings with Connect's directory() middleware

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

var app = connect()
  .use(connect.directory('public'))
  .use(connect.static('public'));

http.createServer(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 }));

http.createServer(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.

11 Deploying Node web applications

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 faced with deploying a web application you'll find yourself considering where to host it. You'll then likely look at 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 then execute the contents of this file. This functionality makes it easy to deploy PHP applications: you simply 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.

Deploying Node applications isn't currently that simple and cheap, however. 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 learn the idiosyncracies of Node-specific cloud hosting services offered by companies like Joyent, Heroku, Microsoft, and Nodejitsu. 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.

Running your own Linux server, however, currently offers the most flexibility because you can easily install any related applications you need, such as database servers. Node-specific cloud hosting currently 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 so you're familiar with setup and maintainance procedures. If you're new to server administration, you can experiment by running software like VirtualBox[1] that 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 skip ahead to section 11.2. If not, we're going to talk about the options available to you: dedicated servers, virtual private servers, and general purpose cloud servers.

11.1.1 Dedicated and virtual private servers

Your server may either be a physical, commonly known as "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 can be administrated 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 pre-existing physical servers.

For web applications that don't anticipate a quick growth in usage, VPSs are a good hosting solution. VPSs are cheap and can be easily allocated additional resources, when needed, such as disk space and RAM. The technology is established and there are many companies, such as Linode[2] and Prgmr[3], that 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, however. Being able to handle quick growth in usage requires the ability to be able 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. They have an advantage over dedicated servers and VPSs, however, 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 cloud hosting automates creation and destruction of an application's servers.

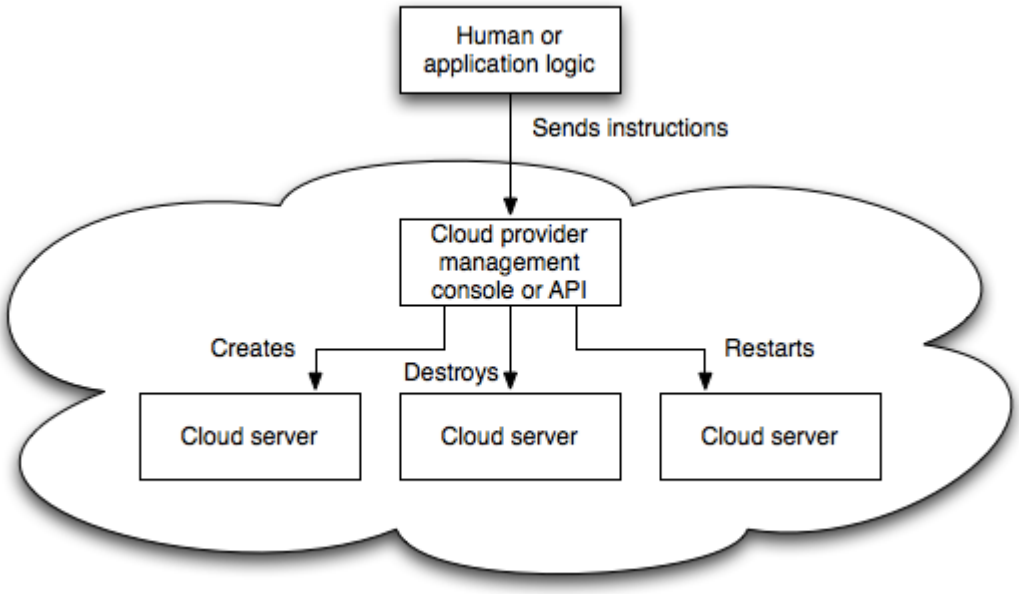

Figure 11.1 Creating, starting, stopping, and destroying cloud servers can be fully automated.

The downside of 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]. Amazon's Elastic Cloud Computing (EC2) service allows you to create servers 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.

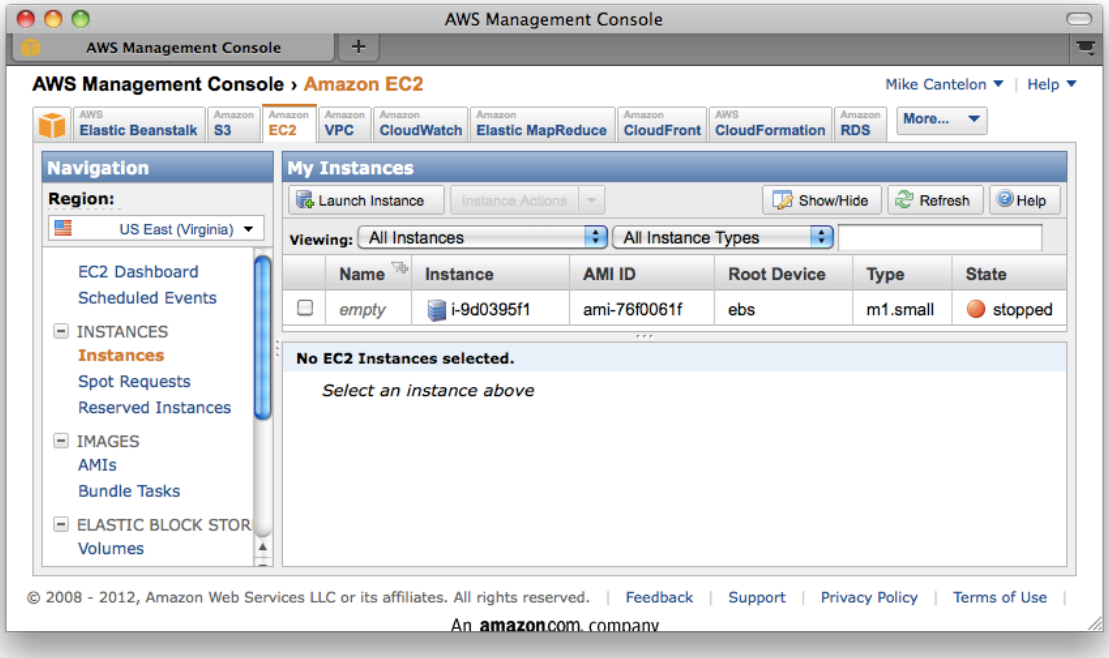

Figure 11.2 The Amazon Web Services web console provides an easier way to manage Amazon cloud servers, for new AWS users, than the command-line.

Luckily, because AWS are so widely used, 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 Cloud[6]. While a gentler learning curve may be appealing, Rackspace Cloud offers a smaller range of cloud-related products and functionality than AWS and has a somewhat clunkier web interface. Rackspace Cloud servers can either be managed using a web interface or community-created command-line tools. Table 11.1 summarizes hosting options.

Footnote 6 http://www.rackspace.com/cloud/

Table 11.1 Summary of hosting options

Suitable traffic growth	Hosting option	Cost
Slow	Dedicated	$$
Linear	Virtual private server	$
Unpredictable	Cloud	$$$

Now that you've got 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 app that you want to show off or maybe you've created a commercial web app and need to test it before putting 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 deployment. Temporary deploys don't persist beyond reboots, but they have the advantage of being quick to set up. In this section, we'll look at a couple of tools helpful for quick deploys.

11.2.1 Deployment from a Git Repository

Before you learn how to deal with various deployment issues, let's run through a naive deployment from 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

Below is an example of starting an application 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.

11.2.2 Keeping Node running

Let's say you've created a personal blog, using the Node-drive Nog blogging application[7], and want to deploy it, making sure that it stays running even if you disconnect from SSH or it crashes.

Footnote 7 https://github.com/c9/nog

The most popular tool for dealing with this is Nodejitsu's Forever. 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.

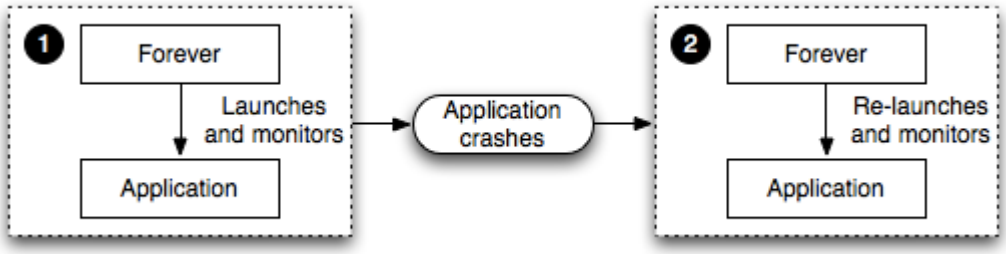

Figure 11.3 The Forever tool helps you keep your application running, even if it crashes.

You can install Forever 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 is this mode, use the `-w` flag.

```
forever -w stop server.js
```

While 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 to ensure - say if you're running your application on a server with a quad-core CPU - that you're not just using 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 then your web application won't be able to consistantly 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 so you'll instead want to use technologies like Nginx that specialize in this.

In this section you'll learn how to use Upstart, a tool included in modern Linux distributions, and Monit, a time-tested tool for restarting crashed applications, to keep your application running as long as the server is. You'll also learn how to use Node's "cluster" API to make sure your application gets the most out of a server's multi-core processor. Last, you'll learn how to use the Nginx server for hosting your Node application's static files.

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 your 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 so you can diagnose any underlying issues.

In Ubuntu and Centos, the two most popular Linux distributions for serving web applications, there are application management and monitoring services that help keep a tight reign 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`. Each file must be set as executable.

The follow steps will create an empty Upstart configuration file for the earlier example application.

```
sudo touch /etc/init/hellonode.conf
sudo chmod u+x /etc/init/hellonode.conf
```

Now for the contents of the configuration file. Put something like the following in your configuration file.

Listing 11.1 A typical Upstart configuration file

```
#!upstart
description "node.js application"

start on startup                                                    (1)
stop on shutdown                                                    (2)

script
    export HOME="/home/username"                                    (3)

    echo $$ > /var/run/yourapp.pid                                  (4)
    exec sudo -u username /usr/local/bin/node /path/to/yourapp.js   (5)
    >> /var/log/yourapp.log 2>&1
end script
```

(1) Make application start when server starts
(2) Make application stop when server stops
(3) Set any environmental variables necessary to the application
(4) Take note of the application's process ID
(5) Run the application using a specific user

If you'd like starts and stops to be logged, you can add the following.

Listing 11.2 Adding logging to an Upstart configuration file

```
pre-start script
    # Date format same as (new Date()).toISOString() for consistency
    echo "[`date -u +%Y-%m-%dT%T.%3NZ`] (sys) Starting"
    >> /var/log/yourapp.log                                         (1)
end script

pre-stop script
    rm /var/run/yourprogram.pid
```

```
    echo "[`date -u +%Y-%m-%dT%T.%3NZ`] (sys) Stopping"
    >> /var/log/yourapp.log                                    (2)
end script
```

(1) Log the start of the application
(2) Log the stopping of the application

Now that you've created an Upstart configuration file you can start your application using the following command.

```
sudo start hellonode
```

The advantage of using Upstart is your application will stop, then automatically restart, when the server is rebooted.

Now that you know how to keep your applications starting and stopping in sync with your servers, lets look at how you can better cope with your applications crashing.

USING MONIT TO PREVENT APPLICATION INTERRUPTION

Monit is a veratile, easy to configure service that keeps an eye on other applications and services. It can keep an eye on pretty much anything. Monit can not only restart crashed applications, it can also send you email alerts.

Enter the following command to install Monit on Ubuntu.

```
sudo apt-get install monit
```

To painlessly install Monit on Centos 6, you'll want to install support for the Repoforge repository to your system. Please visit Repoforge's home page[8] for instructions.

Footnote 8 http://repoforge.org/use/

With Repoforge installed the next step is the actual installation.

```
sudo yum install monit
```

NOTE **Ubuntu**
After installing Monit in Ubuntu you'll need to activate via a quick tweak to its configuration file. This files is found at `/etc/default/monit`. Edit it and change `startup=0` to `startup=1`.

With Monit installed, you'll next want to add a configuration file, similar to the following, for your application. Put this configuration file in the `/etc/monit.d` directory, naming the file after your application: `hellonode.conf` for example. Note the use of `/sbin/start` and `/sbin/stop`, indicating you're managing your Node application's starts and stops with Upstart.

```
check host hellonode.com with address 127.0.0.1
     start "/sbin/start hellonode"
     stop "/sbin/stop hellonode"
     if failed port 8000 protocol HTTP
          request /
          with timeout 5 seconds
          then restart
```

After making this change, enter the following into the command-line to start Monit.

```
/etc/init.d/monit start
```

With the ability to keep your application running regardless of crashes and server reboots taken care of, the next logical concern is performance which Node's cluster API can help with.

11.3.2 Leveraging multiple cores

Most modern computer CPUs have multiple cores. Node applications, however, traditionally only use one of them when running. If you were hosting a Node application on a server and wanted to maximize the server's utilization 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 simultanously using multiple multiple cores. Figure 11.4 visually represents how an application's processing would be organized, using the cluster API on a 4-core processor.

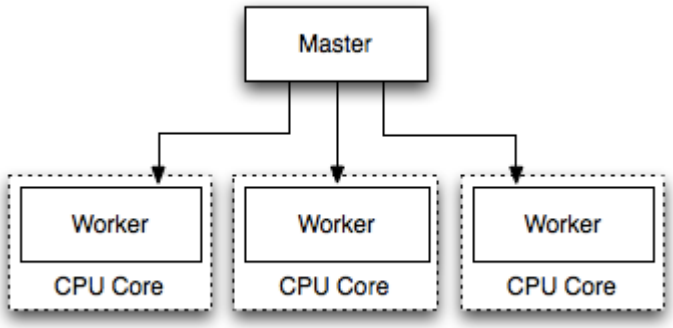

Figure 11.4 A visualization of a master spawning three workers on a four core processor.

The following example automatically spawns a master process and a worker for each additional core.

Listing 11.3 A demonstration of Node's cluster API.

```
var cluster = require('cluster')
  , http = require('http')
  , numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('death', function(worker) {
    console.log('Worker ' + worker.pid + ' died.');
  });
} else {
  http.Server(function(req, res) {
    res.writeHead(200);
    res.end('I am worker ID ' + process.env.NODE_WORKER_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 each**

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

While Node is a very 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 that specific software projects are optimized for, having focussed on this task primarily for many years.

Fortunately, the fastest open source web server in the world, nginx[9], is very easy to use with Node to serve static 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 configure is representated visually in figure 11.5.

Footnote 9 http://nginx.org/

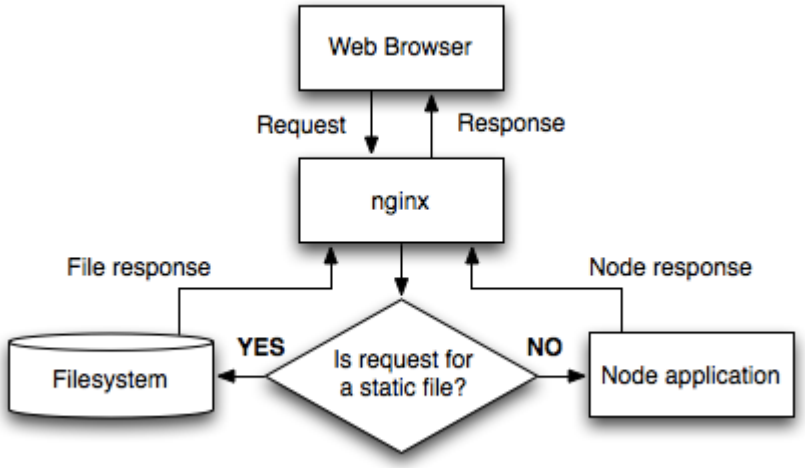

Figure 11.5 nginx can be used as a proxy to relay static assets quickly back to web clients.

The following nginx configuration, which would be put in the nginx configuration's `http` section, implements this setup. The configurate file conventionally stored in a Linux server's `/etc` directory at `/etc/nginx/nginx.conf`.

Listing 11.4 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;
  }
```

❶ **IP and port of Node application**

```
  server {

    listen 80;
    server_name localhost domain.com;
    access_log /var/log/nginx/my_node_app.log;

    location ~ /static/ {
      root /home/node/my_node_app;
      if (!-f $request_filename) {
        return 404;
      }
    }

    location / {
      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;
    }
  }
}
```

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 make sure 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: Node-specific hosting as well as dedicated, virtual private server, and cloud hosting. Each option is suitable for 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. For long-term deployment, however, you may want to automate the starting and stopping of your application using Upstart and supervise the running of your application using Monit.

To get the most of your server resources, Node's cluster API can be leverages to run application instances simultaneously on multiple cores. You may also, if your web application requires serving static assets such as images and PDF documents, want to run the Nginx server and proxy your Node application through it.

Having gotten a good handle on the ins-and-outs 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 other applications of Node: everything from building command-line tools to "scraping" data from websites.

13 The Node ecosystem

In this chapter

- Where to go online to get help with Node
- How to collaborate on Node development using GitHub
- Publishing your work using the Node Package Manager

In order to get the most out of Node development, it helps to understand where to go for help and how to share your contributions with the rest of the community. If you've been using Node for awhile and are familiar with how things work, you may want to skip this chapter but, otherwise, read on.

As with most open source communities, development of the Node framework and related projects happens via online collaboration. Many people come together to submit and review code, document projects, and report bugs. When a new version of the Node framework is released 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. Support for the Node framework and related projects is provided through online resources. Figure 12.1 shows how online resources are used for Node-related development, distribution, and support.

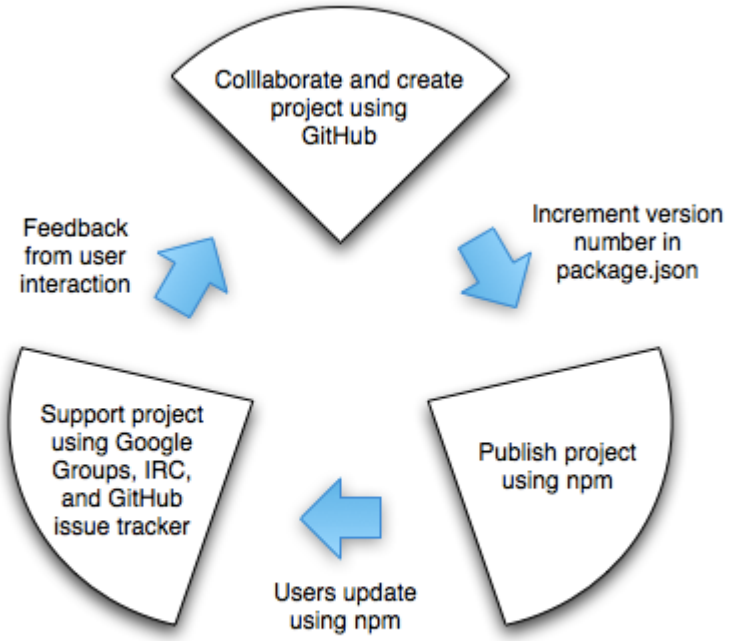

Figure 13.1 Node-related projects are created via online collaboration, often via the GitHub website. They are then published to npm with documentation and support provided through online resources.

As you'll likely need support before needing to collaborate, let's first look at where to go to get help when you need it using online resources.

13.1 Online resources for Node developers

As the world of Node is ever-changing, most references are online. There are websites, online discussion groups, and chat rooms where you can go to get the information you need.

13.1.1 Node and Module References

Table 12.1 lists a number of useful Node-related online references and resources. The most useful websites for referencing the Node APIs and learning about available third-party modules are nodejs.org, shown in figure 12.2, and npmjs.org, respectively.

Table 13.1 Useful Node.js references

Resource	URL
Node.js home page	http://nodejs.org/
Most recent Node.js core documentation	http://nodejs.org/docs/latest/api/
Node.js blog	http://blog.nodejs.org/
Node.js job board	http://jobs.nodejs.org/
Node Package Manager home page	http://npmjs.org/
Node Package Manager search page	http://search.npmjs.org/

When attempting to implement something using Node, or any of its built-in modules, the Node home page is an invaluable resource. The site documents the entirety of the Node framework, including each of its APIs. Documentation for the most recent version of Node can always be found on the site. The official blog documents the latest Node advances and shares important community news. There's even a job board available.

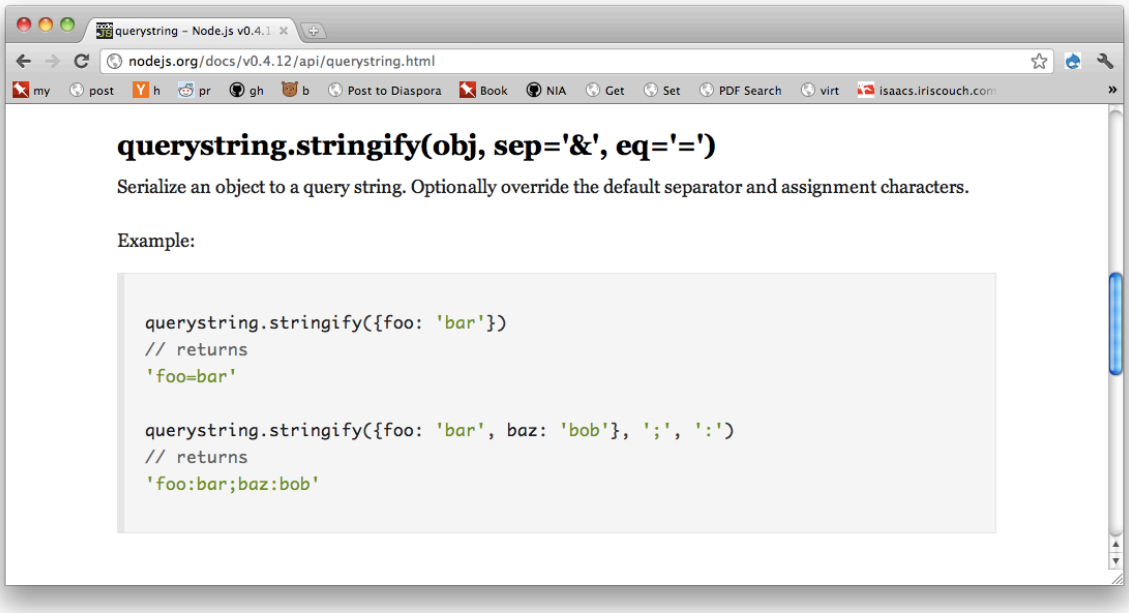

Figure 13.2 As well as providing links to useful Node-related resources, Nodejs.org provides definitive API documentation for every version of Node ever released.

When shopping for third-party functionality, the npm repository search page is the place to go. It allow you to search through the thousands of modules available in npm using keywords. If you find a module that looks interesting, clicking on the module's name brings you to the detail page, which provides a links to the project's 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 package is compatible with, and how the module is licensed.

If you have a question about how Node or third-party modules are used, however, these websites may not help. Luckily there are some great places to ask for help online.

13.1.2 Google groups

Google Groups exist for Node and for some popular third-party modules such as npm, Express, node-mongodb-native, and Mongoose. Google groups are great for tough or in-depth questions. If you, for example, 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 Group[1] and search it to see if anyone else had the same problem. If noone has had the same problem, then the next step would be to join the Google Group and post your question. A Google Groups post can be quite long, so it your question is complicated you have the opportunity to explain it thoroughly.

Footnote 1 http://groups.google.com/group/node-mongodb-native

There is no central list of all Node-related Google Groups, however. In order to find them they'll either be mentioned in project documentation or you'll have to search the web to find out. You could, for example, Google "nameofsomemodule node.js google group" to check if a Google Group existed for a third-party module.

The drawback to Google Groups, however, is that you usually have to wait anywhere from hours to days to get a reponse, depending on the Google Group. For small questions where you want a quick response, going to an Internet chat room can offer a quicker solution.

13.1.3 IRC

Internet Relay Chat (IRC) was created way back in 1988 and is thought by some to be archaic, but it's still very much alive - the best way, in fact, to get quick questions about open source software answered online. IRC rooms are called "channels" and exist for Node and various third-party modules. There's no list of Node-related IRC channels everywhere but third-party modules that have them will sometimes mention them in their documentation.

To get your question answered on IRC you simply connect to an IRC network, change to the appropriate channel, then 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 by a quick web search.

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 enter a chatroom, enter the appropriate channel into the connection form. You don't need to register and you can enter any nickname - if someone is already using it the character '_' will be appended to the end of your nickname to differentiate you.

Once you click "Connect" you'll should end up in a chat room with a list, in the right sidebar, of other users in the same room.

13.1.4 GitHub Issues

Another place to look for solutions to problems with Node modules is, if the project's development occurs on GitHub, 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 then use the search form to search for issues related to your problem. An example issue queue is shown in figure 12.3.

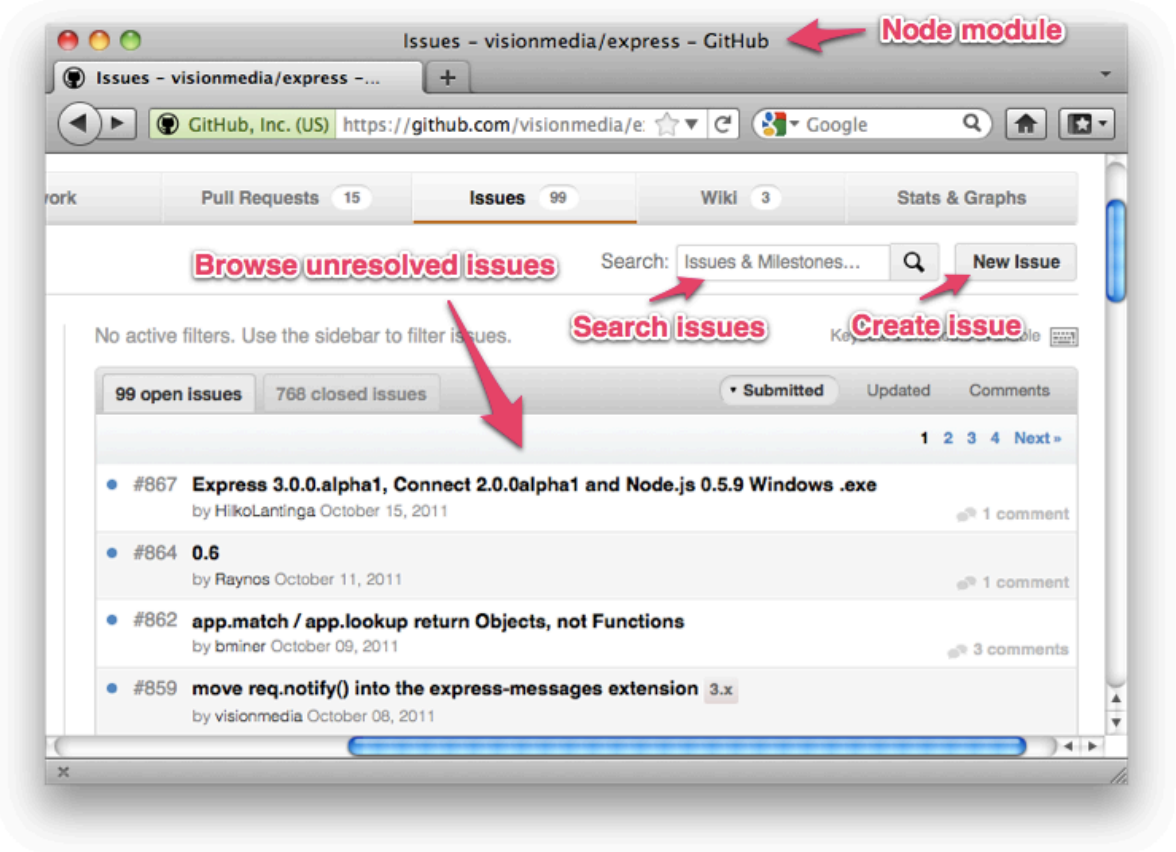

Figure 13.3 For projects hosted on GitHub, the issue queue can be helpful if you've been otherwise unable to solve a development problem relating to the project.

If you are unable to find an issue that addresses your problem, and think the problem is due to a bug in the system, you can click "New Issue" on the issues page to describe your problem. Once you've created an issue project maintainers will be able to reply on the page for that issue offering solutions or asking questions to get a better idea of your problem.

Now that you've got an idea where to go online for general help, 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. GitHub is a service that provides hosting for Git, a powerful version control system (VCS), and a web interface that allows Git repositories to be easily browsed. GitHub usage is free for open source projects.

NOTE **Git**

The Git VCS has become a favorite among open source projects. It is a distributed version control system (DVCS) so it, unlike Subversion and many other VCSs, can be used without a network connection to a server. Git was released 6 years ago, inspired by a proprietary VCS called BitKeeper. The publisher of BitKeeper had granted the Linux kernel development team free use of the software, but revoked it due to the suspicion 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 a great help in 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 simply "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 guage a project's overall popularity.

GitHub is powerful, but how do you use it?

13.2.1 Getting started on GitHub

Once you've come up with an idea for a Node-based project or third-party module you'll want to set up an account on GitHub, if you haven't already, for easy access to Git hosting. Only after you're set up can you add projects, which you'll learn to do in the next section.

As GitHub requires use of Git, you'll want to configure Git before continuing to GitHub. Once Git is configured, getting set up on GitHub involves registering on the GitHub website and providing a Secure Shell (SSH) public key. The SSH key is needed to keep your interactions with GitHub secure. You'll learn the details of each of these steps in this section. Note that you only have to do these steps once, not everytime you add a project to GitHub.

GIT CONFIGURATION AND GITHUB REGISTRATION

Usage of GitHub requires that you configure your Git tool, letting it know your name and email address. Enter the following two commands to do so.

```
git config --global user.name "Bob Dobbs"
git config --global user.email subgenius@example.com
```

Next, register on the GitHub website. Visit their sign-up form[2], fill in the form, then click "Create an account".

Footnote 2 https://github.com/signup/free

PROVIDING GITHUB WITH AN SSH PUBLIC KEY

Once you're registered, the second thing you'll need to do is provide GitHub with an SSH public key[3]. This will be used to authenticate your Git transactions. To do so:

Footnote 3 http://github.com/guides/providing-your-ssh-key

1. Click "Account Settings"
2. Click "SSH Public Keys"
3. Click "Add another public key"

At this point what you need to do varies depending on your operating system. GitHub will detect your operating system and show relevant instructions.

Now that you're set up to use GitHub, lets move on to how to put your projects on GitHub.

13.2.2 Adding a project to GitHub

Now that you're set up on GitHub, you can add a project to it. To do so, you'll create a GitHub repository for your project, create a Git repository on your workstation, then send the work you've done on your local repository to the GitHub repository. Figure 12.4 outlines the process, the steps for which we'll walk you through. Your project files will then be viewable using GitHub's web interface.

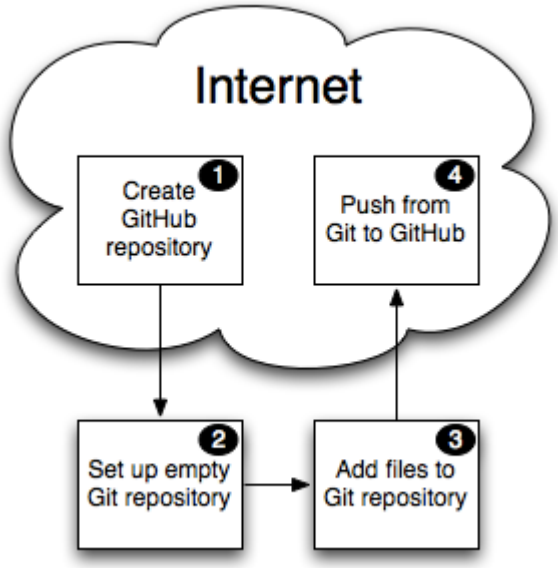

Figure 13.4 The steps involved in adding a Node project to GitHub

CREATING A GITHUB REPOSITORY

To create a repository on GitHub, log in, click on "Dashboard", then click on "New Repository". A GitHub repository creates an empty Git repository and issue queue for your project. Fill out the resulting form describing your repository and click "Create Repository". You'll then be presented with the steps needed to use Git to push your code to GitHub. As it's helpful to understand what each of these steps is doing, in this section we'll run through an example of these steps and you'll learn the bare essentials of Git usage.

SETTING UP AN EMPTY GIT REPOSITORY

As an example project to add 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. This will create a directory called ".git" that contains repository metadata.

```
git init
```

ADDING FILES TO A GIT REPOSITORY

Now that you've got an empty repository set up, 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 content into a file in your repository directory called "index.js".

Listing 13.1 index.js A Node module for URL shortening.

```
exports.initPathData = function(pathData) {
  pathData = (pathData) ? pathData : {};
  pathData.count = (pathData.count) ? pathData.count : 0;
  pathData.map   = (pathData.map) ? pathData.map : {};
}

exports.shorten = function(pathData, path) {
  exports.initPathData(pathData);
  pathData.count++;
  pathData.map[pathData.count] = path;
  return pathData.count.toString(36);
}

exports.expand = function(pathData, shortened) {
  exports.initPathData(pathData);
  var pathIndex = parseInt(shortened, 36);
  return pathData.map[pathIndex];
}
```

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

Note that if your workstation was struck by lightning now you'd lose all your work. To safeguard against this, 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. Before doing this, however, 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 simply as "remotes". The following line shows how you add a GitHub remote to your repository. Substitute "username" with your username. "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. Sending changes is called, in Git terminolagy, a repository "push". In the following command, tell Git to push your work to the "origin" remote that 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 listed.

Creating a module and hosting it on GitHub is a quick-and-dirty way to be able to reuse it. If you wanted, for example, to use your sample module in a project, you could enter the following commands. `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: 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 embracing of Git, there are many excellent resources out there that will 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 online[4]. If a hands-on approach is more your style, the official Git site's documentation page[5] lists a number of tutorials that will get you up and running.

Footnote 4 http://progit.org/

Footnote 5 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.

Lets say you're using a third-party module and run into a bug in it. You may be able to examine the module's source code and figure out a way to fix the bug. If so, you could email the author of the code, describing your fix and attaching files containing your fixes. This would require the author to do some tedious work, however. The author would have to compare your files to her own and combine the fixes from your files with the latest code. If the author was using GitHub, however, 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, if you wanted to, you could continue to maintain and enhance independently. And, if your fork grew in popularity, somebody might very well fork your fork then 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 simply 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 12.5 shows, visually, an example GitHub collaboration scenario.

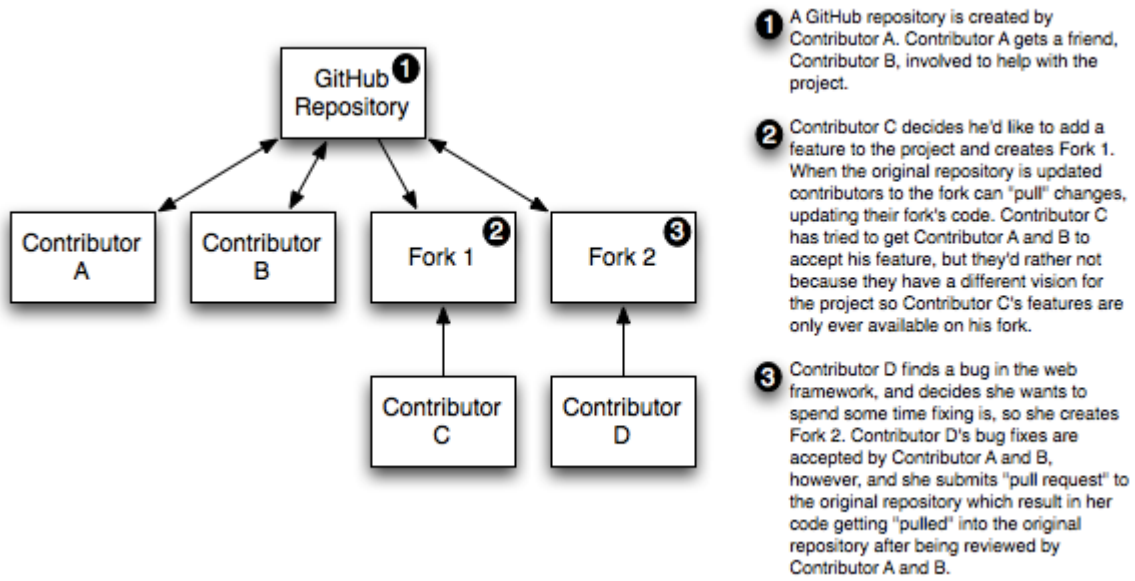

Figure 13.5 An example of a typical GitHub development scenario

Now, let's walk through an example, represented visually in figure 12.6, of forking a GutHub repository for the purpose of collaboration. Forking starts the collaboration process by duplicating the repository on GitHub. You then clone the forked repository to your workstation, make changes to it, commit the changes, push your work back to GitHub, then send a pull request to the owner of the original repository asking them to consider your changes. If they want to include your changes in their repository they then approve your pull request.

Figure 13.6 The process of collaborating on GitHub via forking.

Let's say you want to fork the `node-elf` repository created earlier and add to the URL shortening module so the module's version number is specified by the module logic itself. This would allow anyone using the module to make sure they are 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 result is a page similar to the original repository page with "forked from mcantelon/node-elf" displayed under the repository name.

After forking, the next step is cloning the repository to your workstation, making changes, then pushing the changes to GitHub. Carrying on with the example, the following commands will do this 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. Then enter the subject and body of a message describing your changes then click "Send pull request". You will then see a screen containing content similar to figure 12.7.

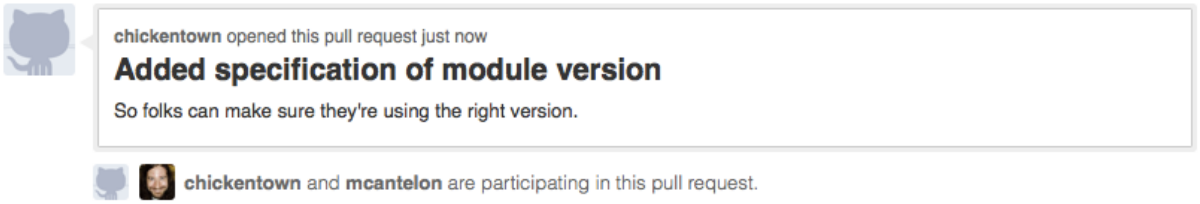

Figure 13.7 The details of a GitHub pull request.

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 simply 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 it 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 awhile 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. You'd be limited in the number of Node users you'd reach, however, and once people started using your module you wouldn't have a way to let them know of updates to your module.

Publishing to npm, however, solves the problems of discoverability and providing updates. In addition, 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. For another example, a module that provides a command-line tool might have as a dependency a helper module for parsing command-line arguments.

Up until now 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 subsection we'll show you each step needed to publish your own work on npm: preparing a package, writing a package specification, testing a package, and publishing.

13.3.1 Preparing a package

Any Node module that you wish 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 a quick start.

The packge directory should be organized using subdirectories. Conventional subdirectories are "bin", "docs", "example", "lib", and "test" and their usage is listed in table 12.2.

Table 13.2 Conventional subdirectories in a Node project

Directory	Usage
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 publishing a package to npm, you need to including 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 website[6] that shows a sample package.json file and explains what each part of the sample file is for when you move your mouse over it.

Footnote 6 http://package.json.nodejitsu.com/

In a package.json file, only the name and version are manditory. 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.

Here's what a sample spec 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
```

As generating JSON by hand is only slightly more fun than hand-coding XML, there are tools that make it easier. ngen, for example, is an npm package that, if installed, adds a command-line tool called "ngen". ngen will generate a package.json file, after asking a number of questions. It will 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 running ngen in the root directory of a module you'd like to publish to npm, you'll end up with output like the following. Some files may be generated that you don't need. If so, just 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 done this, you can move on to actually publishing your module.

13.3.3 Testing and publishing a package

Publishing a module to npm involves three steps, shown visually in figure 12.8: testing installation of your package locally, adding an npm user if you haven't already, and, finally, the actual publishing.

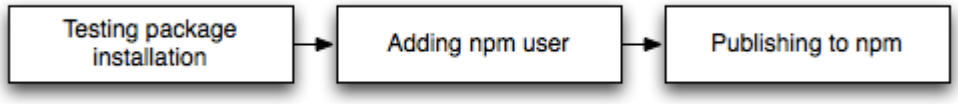

Figure 13.8 The process of publishing a package in the npm repository.

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, so it can be used by Node just like a package conventionally installed by npm.

```
sudo npm link
```

Once you've done this you can create a test directory in which you can link-install the package to test it.

```
npm link elf
```

Once 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 shows. You should see, as a result, 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 you can then, if you've finished development of 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.

ADDING AN NPM USER

Enter the following to create an npm publishing account for yourself.

```
npm adduser
```

You'll be prompted for a username, email, and a password. If the account gets added successfully, no error will be shown.

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 insecure channel", but if you don't see additional errors them your module was likely 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 repos as npm package dependencies, you can. Perhaps you have a module of useful helper functions that you'd like to use, but not release publically on npm. To add a private dependency, where you would normally put the dependency module's name, you can put any name different from 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 repo.

```
"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 module, set the "private" characteristic in its package.json file to "true".

Now that you know how to get help, collaborate, and contribute you are able to dive right into the Node community.

13.4 Summary

As with most successful open source projects, Node has an active online community. Because of this online resources are plenty and questions can usually be quickly answered 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 respository 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. Changes made to a fork can be easily submitted back to the original repostory.

Once a Node project has reached the stage where it's worth sharing with the world at large, it can be submitted to the Node Package Manager repository. Inclusion in npm makes your project easier to find for others and, if your project is a module, inclusion in npm means your module will be easy to install.

Now that you know how to get the help you need, collaborate online, and share your work you're ready to get the most out of Node development.

A Installing Node on Windows using Cygwin

While Cygwin isn't generally the best way to run Node in Windows, it offers a layer of Unix compatibility that makes it works with some modules that won't work in Windows natively.

To install Cygwin, navigate to the Cygwin installer download link[1] 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 1 http://www.cygwin.com/setup.exe

You should now see Cygwin's package selector, as shown in figure A.1. 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).

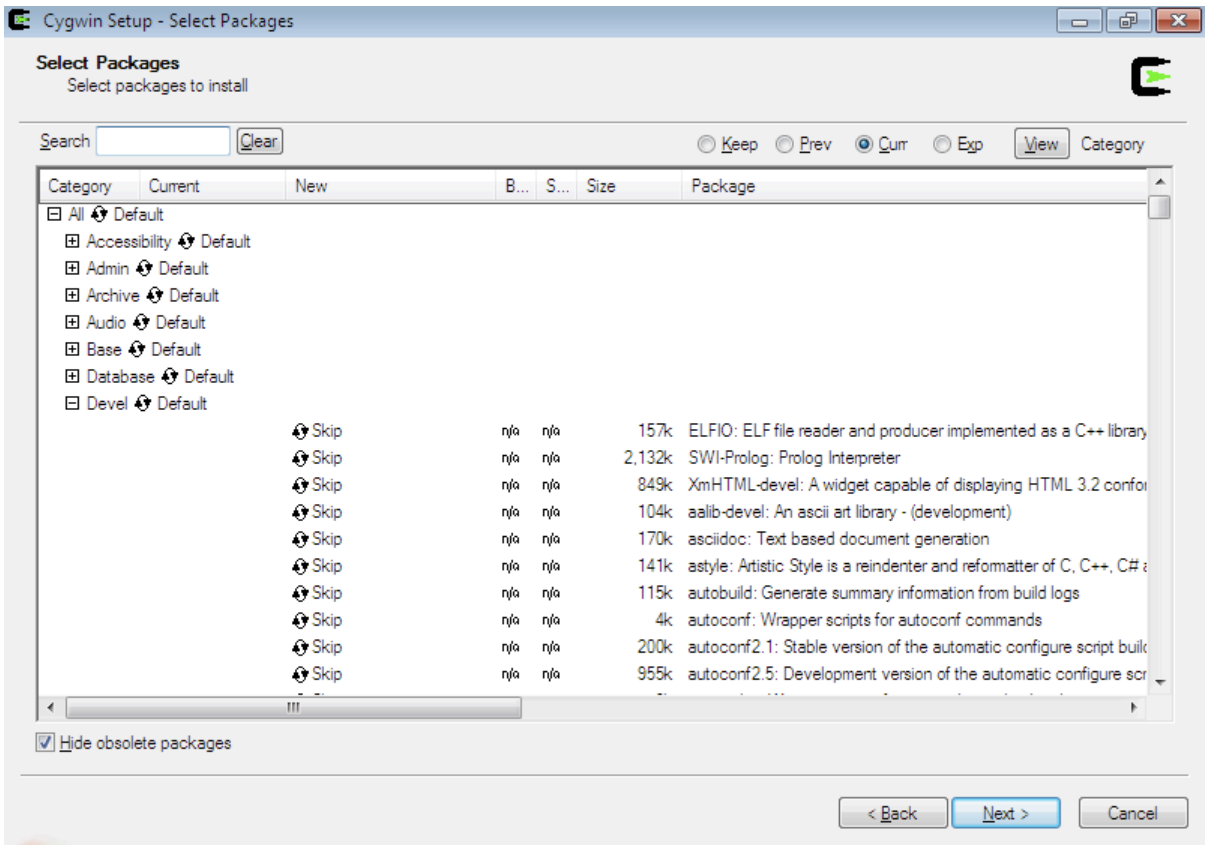

Figure A.1 The Cygwin's package selector allows you to select open source software that will be installed on your system.

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 2.2.4 "Compiling Node", in chapter 2, for the necessary steps).