Node.js IN ACTION

Mike Cantelon
TJ Holowaychuk

MANNING

MEAP Edition
Manning Early Access Program
Node.js in Action version 4

Table of contents

1 Why the Web needs Node

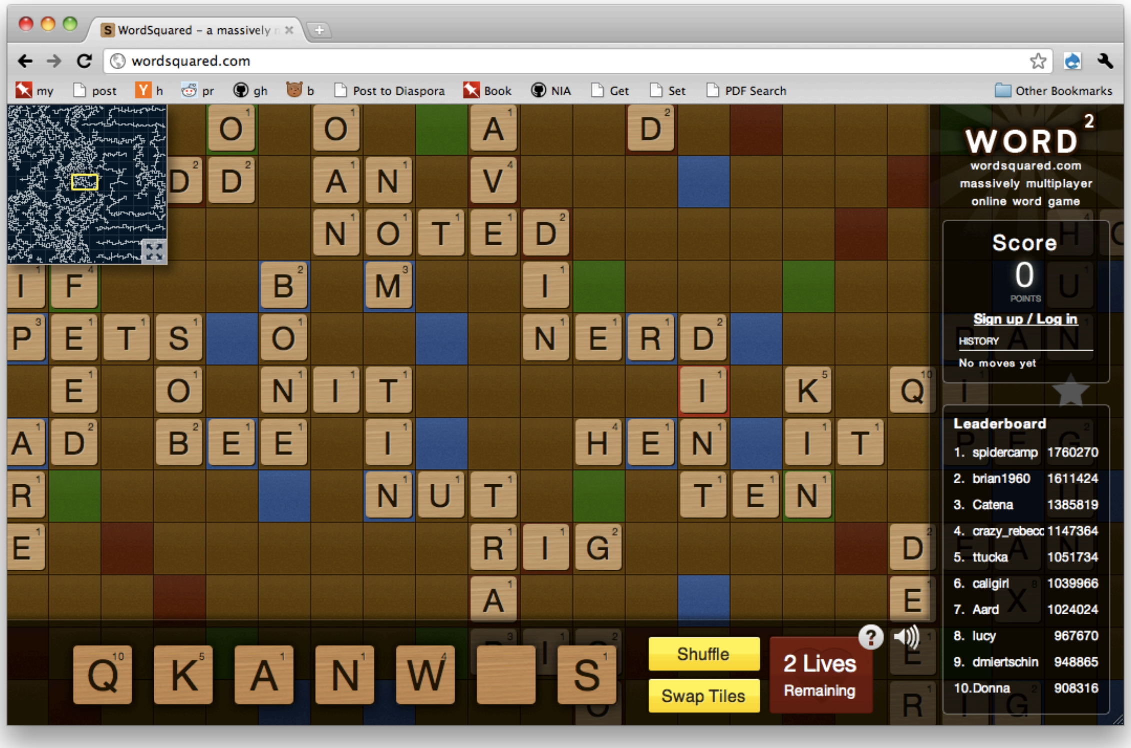
Imagine if web apps could change in real time to provide information instantly: if you could see a friend's email as she types it, if you could see the bus approaching on your Smartphone web browser's map, if you could play large multiplayer arcade games with nothing but a web browser.

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, 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.



Leaderboard

1.	spidercamp	1760270
2.	brian1960	1611424
3.	Catena	1385819
4.	crazy_rebecc	1147364
5.	ttucka	1051734
6.	caligirl	1039966
7.	Aard	1024024
8.	lucy	967670
9.	dmiertschin	948865
10.	Donna	908316

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 messenging). 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.

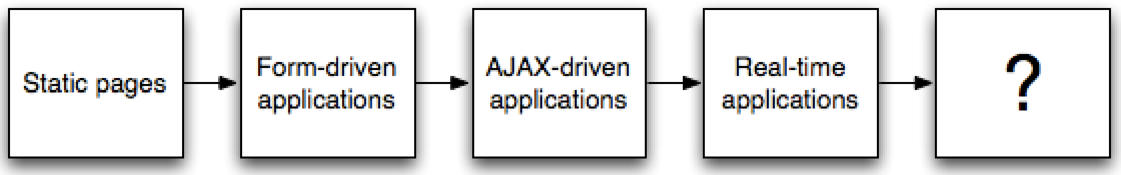


Figure 1.2 A short history of Web technology. As immediacy of interactivity increases, web pages become more like traditional desktop applications that don't require installation. With Node making real-time interactivity accessible, who knows where the Web will go next?

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.

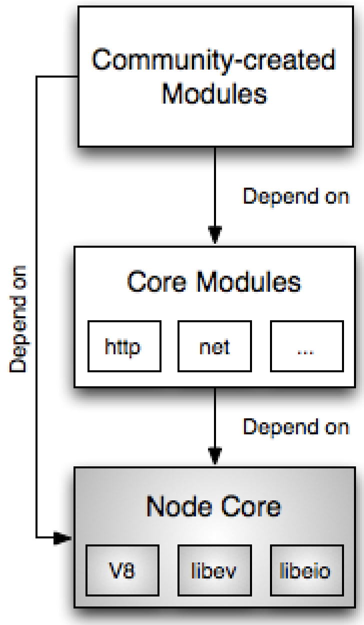


Figure 1.3 Node is conceptually composed of three layers of functionality: the underlying core engine, a number of core modules that add provide utility functions and APIs, and community-created modules for everything else.

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.

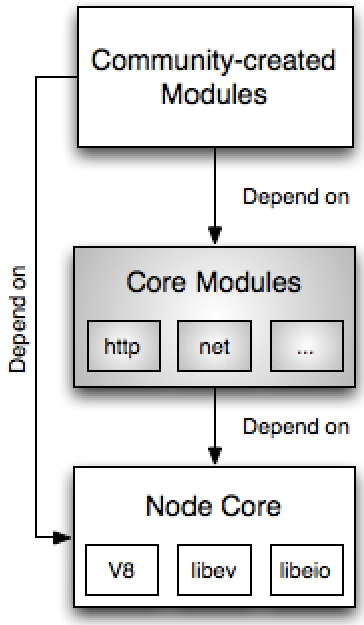


Figure 1.4 Node's core modules provide a wide range of functionality.

NOTE **UDP**
UDP is an Internet networking protocal, supported by Node's networking API, that allows data to be sent quickly, although without any feedback on whether or not the data reaches its destination. The Domain Name System (DNS) system is one example of a networking application that uses this protocol.

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 community-contributed server management applications like Cluster[1] make it easy.

Footnote 1 http://github.com/learnboost/cluster

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. 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.

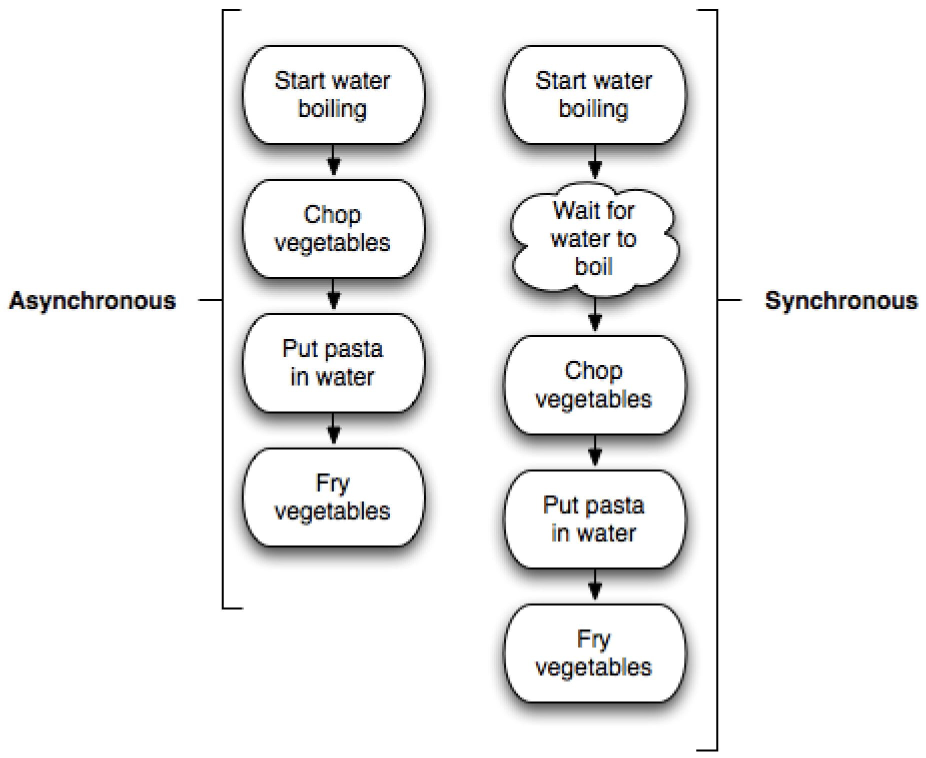


Figure 1.5 Using cooking as an example, asynchronous execution is shown to be more efficient because it requires less time be spent waiting around. Node-based asynchronous development results in efficient applications.

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 Using multiple callback arguments in a function

```
function horse_race(success, failure) {

  var victory = Math.round(Math.random(1));

  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, as a result, improve programmer productivity. Dynamic languages don't require compilation, 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"[2] 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 2
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 further differentiates itself from established web frameworks in its capability of going beyond HTTP. 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 even be used as a reverse proxy, caching web requests from a slower, non-Node web server (potentially usurping the role of special-purpose applications like Squid and Varnish).

Node is also a good fit for designing designing protocols and command-line applications. But before we get into those topics, let's look at how you can use Node to design server applications

1.3.1 Server applications

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.

Node versions of a variety of traditional server applications have already been written. Node-based DNS servers, web crawlers, message queue servers, and many other types of applications exist.

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.

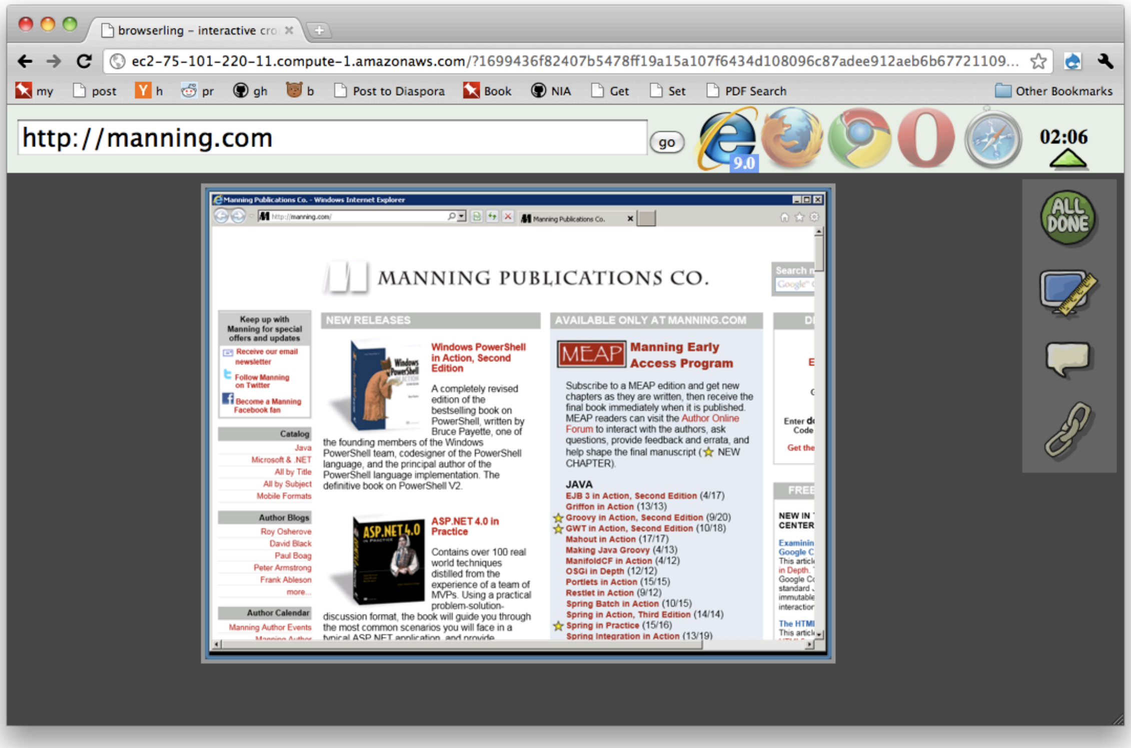

browserling – interactive cro
ec2-75-101-220-11.compute-1.amazonaws.com/?1699436f82407b5478ff19a15a107f6434d108096c87adee912aeb6b67721109...
http://manning.com
go
9.0
02:06
ALL DONE
MANNING PUBLICATIONS CO.
NEW RELEASES
AVAILABLE ONLY AT MANNING.COM
Windows PowerShell in Action, Second Edition
ASP.NET 4.0 in Practice
MEAP Manning Early Access Program

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 would be nothing without protocols and, in addition to the shift towards a real-time Web, another Node trend is the development of new protocols that deal with some of the Web's current challenges. 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.

Another area in which there is interest in developing new protocols, and where Node will come into play, is the realm of social networking. The reason behind this is concern that large social network sites are negatively changing the character of the Web.

Until now, the web has achieved its large social value by presenting information in a technically unrestricted manner using non-proprietary standards like HTML, CSS, and RSS. These standards have allowed sites to freely link to each other, has made the barrier to setting up new sites low, and has allowed search engines to index the Web's content. The semantic web, which some envision as part of the Web's future, is reliant on the idea that websites will expose their data in a structured way that can be consumed by web crawlers and run through algorithms, which are then able to add value to the web by indexing and analyzing this data.

The rise of social networking sites like FaceBook, however, has given rise to a different vision of the Web. These sites gain value through user-created information but often don't allow users to export the information they've created or maintain control of exactly how the information they add is used. These sites are also easily able to censor information, perform analysis to find out who knows who, or provide private information to state authorities.

In response, there is a growing movement towards the creation of protocols and distribution mechanisms that will allow users to enjoy social networking functionality freely. While there are a number of projects tackling this issue, it's not yet a solved problem. Node, which provides an accessible means to develop low-level TCP/IP applications, seems ideally suited to the challenge.

In fact, Node is called "Node" in part because its creator envisioned servers created in Node working together in a network, either locally, over a WAN, or over the Internet. If you're a die-hard dreaming about 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 Unix command-line interface

In addition to refreshing your Javascript knowledge, it's also worth making sure you've got a good grasp of the Unix 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). By "Unix" we don't mean any specific type of Unix system, but any operating system that supports the Portable Operating System Interface for Unix (POSIX). This includes Mac OS X, Linux, and Cygwin (a Unix-like environment that is run in Microsoft Windows).

Knowing the most used Unix commands is useful because 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. 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.

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 open source Cygwin project provides a Unix-like CLI and is available online for free download. 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

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. One of the first thing you'll need to do to start when starting Node development is using the command-line to install Node itself.

2.2 Installing Node

Unlike a lot of software, Node is usually installed via the command-line. Conventional application installation involves clicking and selecting things in a graphic user interface, whereas command-line installation involves entering commands. Node command-line installation involves compiling from source code which ensures that your Node installation is fresh, incorporating the latest fixes. Command-line installation can take some getting used to if you're new to it, but the process will become familar during your journey as a Node developer.

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 two most popular ways of installing Node on this operating system are by using a tool called Homebrew (which automates installation from source) or by manually installing from source. For both of these methods you'll need Apple's XCode developer tools.

NOTE **XCode**

If you don't have XCode installed, you can download XCode from Apple's website[3] (note that XCode is a large download: approximately 4GB).

Footnote 3 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
```

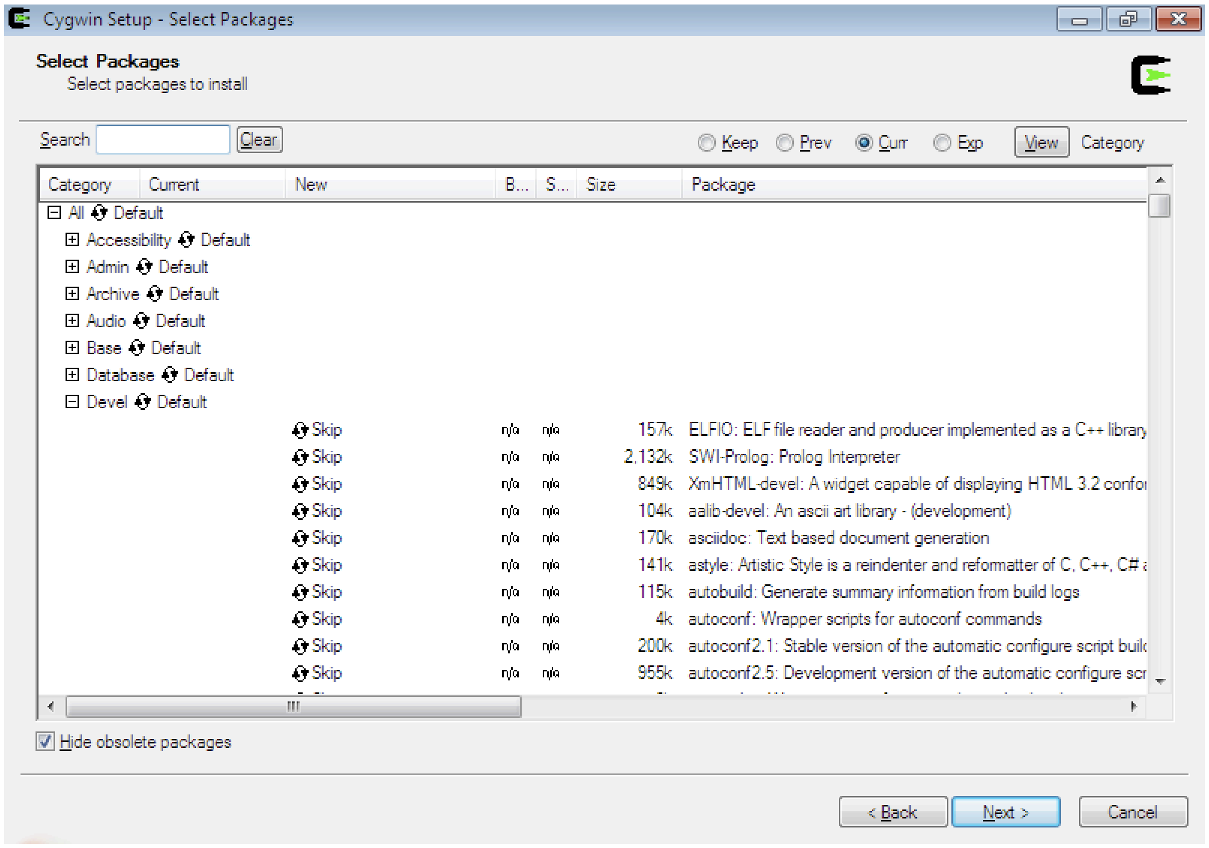
As Hombrew compiles code you'll see a lot of text scroll by. The text is information related to the compiling process and can be ignored.

2.2.2 Windows setup

Running Node in Windows requires the use of a project called Cygwin that provides a Unix-like environment. To install Cygwin, navigate to the Cygwin installer download link[4] 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 4 http://www.cygwin.com/setup.exe

You should now see Cygwin's package selector, as shown in figure 2.1. You'll use this selector to pick what software functionality you'd like installed in your Unix-like environment (see table 2.1 for a list of Node development-related packages to install).



Category	Current	New	B...	S...	Size	Package
⊟ All Default						
⊞ Accessibility Default						
⊞ Admin Default						
⊞ Archive Default						
⊞ Audio Default						
⊞ Base Default						
⊞ Database Default						
⊟ Devel Default						
		Skip	n/a	n/a	157k	ELFIO: ELF file reader and producer implemented as a C++ librar
		Skip	n/a	n/a	2,132k	SWI-Prolog: Prolog Interpreter
		Skip	n/a	n/a	849k	XmHTML-devel: A widget capable of displaying HTML 3.2 confo
		Skip	n/a	n/a	104k	aalib-devel: An ascii art library - (development)
		Skip	n/a	n/a	170k	asciidoc: Text based document generation
		Skip	n/a	n/a	141k	astyle: Artistic Style is a reindenter and reformatter of C, C++, C#
		Skip	n/a	n/a	115k	autobuild: Generate summary information from build logs
		Skip	n/a	n/a	4k	autoconf: Wrapper scripts for autoconf commands
		Skip	n/a	n/a	200k	autoconf2.1: Stable version of the automatic configure script buil
		Skip	n/a	n/a	955k	autoconf2.5: Development version of the automatic configure scr

☑ Hide obsolete packages

Figure 2.1 The Cygwin's package selector allows you to select open source software that will be installed on your system.

Table 2.1 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", later in this section, for the necessary steps).

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.

The first step is to find an archive containing a recent release of the source code. This is located at Node's website[5]. Visit this web page then right-click (CTRL-click if on Mac) on the archive filename (which should end in ".tar.gz"). Next click "Copy Link Address" (you don't need to download it using the browser: you'll use a command-line utility to download it).

Footnote 5 http://nodejs.org/#download

Once in the command-line, you 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 type "wget ", then paste in the link address you copied from the browser earlier and hit ENTER. To paste, press COMMAND-v in OS X, CTRL-v in Ubuntu, or, if in Cygwin, click the Cygwin icon in the upper-left corner of the window, select Edit, then select Paste. You should end up with something like the following:

```
wget http://nodejs.org/dist/node-v0.4.7.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-v0.4.7.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:

```
ls
```

Next, you'll see listed a directory called something like "node-v0.4.7" (again, the "0.4.7" part will likely be different). Enter the following command, substituting the directory name listed, to move into this directory:

```
cd node-v0.4.7
```

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 inssue 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 Telnet 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 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 "hello_telnet.js", into the directory in which your CLI normally starts:

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

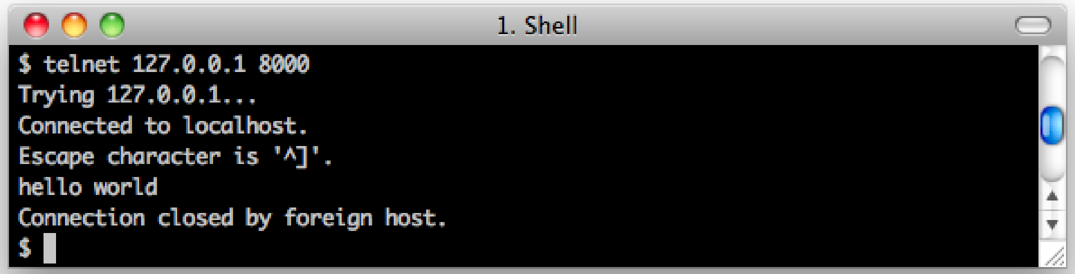
To execute this script, open up a command-line and enter the following:

```
node hello_telnet.js
```

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

```
telnet 127.0.0.1 8000
```

As shown in figure 2.2, you should see, in amongst the text indicating the Telnet command's status, the text "hello world" returned.



```
$ telnet 127.0.0.1 8000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello world
Connection closed by foreign host.
$
```

Figure 2.2 Results of connecting to our "Hello Telnet" server. Note that there is a lot of extraneous text returned that is normally generated when using Telnet rather than returned by the Node script itself.

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 six 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 "server.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 server.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.

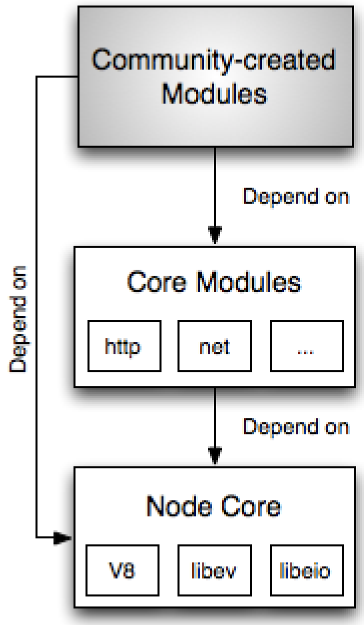


Figure 2.3 The Node stack is composed of globally

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.

Npm requires that Node and the Git version control system be installed. See table 2.2 for platform-specific Git information on installing Git.

Although anyone with an internet-accessible server can set up their own Git version control server, many npm packages use the online service Github[6]. In addition to Git hosting, Github provides issue tracking and hosting of documentation. Visiting an npm package's Github repository is a great way to learn about the package as it allows you to browse source code and get a sense of how many people use the package (Github users often "watch" other repositories to receive updates whenever changes are made: the total number of watchers is a good metric for project popularity).

Footnote 6 http://github.com

Table 2.2 Git Platform Notes

Platform	Instructions
Mac OS X	Modern versions of XCode install Git by default. For those using older versions of XCode, Git is installed, using Homebrew, by opening a CLI and entering`brew install git`. For those not using Homebrew, the Rudix project provides a simple downloadable installer[7].
Cygwin (Windows)	If you've followed our instructions on installing Node in Cygwin, Git should already be installed.
Ubuntu	To install Git open a CLI and enter `sudo apt-get install git`.
Centos	To install Git open a CLI and enter `sudo yum install git`.

Once you've installed both Node and Git, open a command-line and enter the following to install npm.

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

Footnote 8 http://github.com/isaacs/npm

2.4.2 Searching for packages

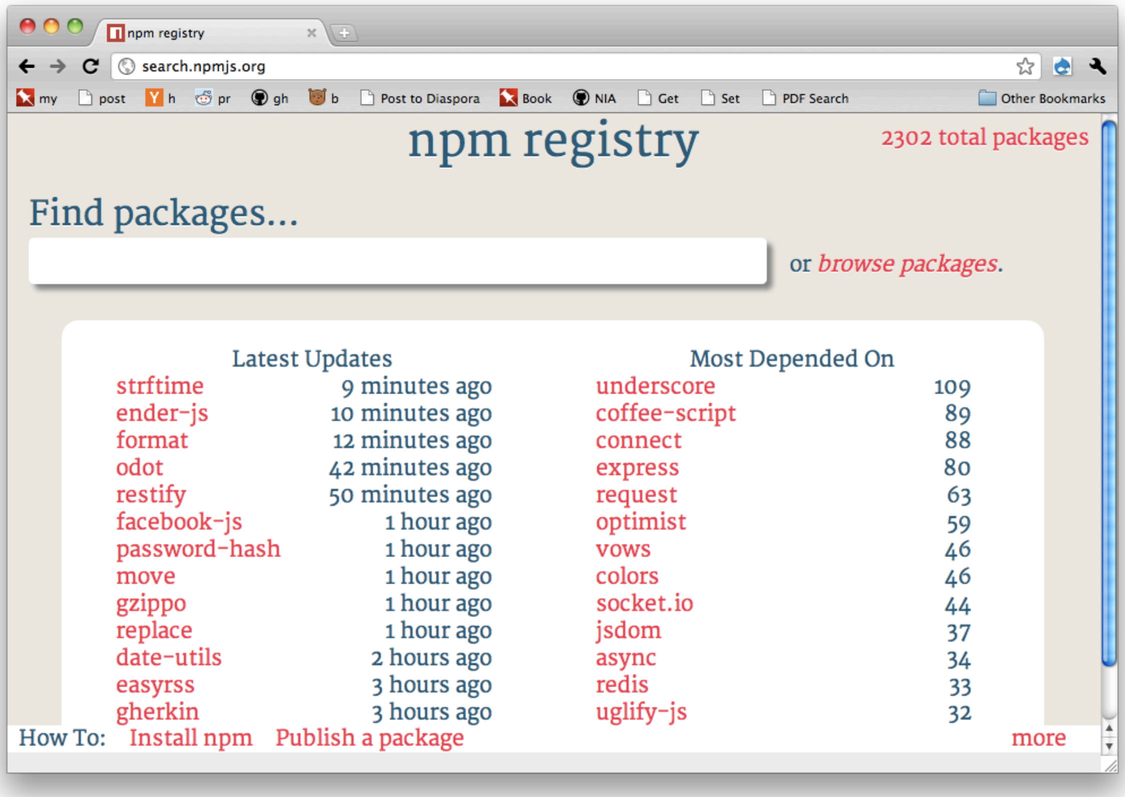
Once npm is installed, you can use npm's `search` command to find packages available in its repository. For example, if you wanted to search for an XML generator, you could simply enter the command:

```
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[9] 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 9 http://search.npmjs.org/



npm registry

2302 total packages

Find packages...

or *browse packages*.

Latest Updates		Most Depended On	
strftime	9 minutes ago	underscore	109
ender-js	10 minutes ago	coffee-script	89
format	12 minutes ago	connect	88
odot	42 minutes ago	express	80
restify	50 minutes ago	request	63
facebook-js	1 hour ago	optimist	59
password-hash	1 hour ago	vows	46
move	1 hour ago	colors	46
gzippo	1 hour ago	socket.io	44
replace	1 hour ago	jsdom	37
date-utils	2 hours ago	async	34
easyrss	3 hours ago	redis	33
gherkin	3 hours ago	uglify-js	32

How To: Install npm Publish a package more

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 by default, a directory traditionally used by Unix to store user installed applications.

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 is to group related logic into separate files.

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. Node modules bundle up code for reuse, but don't alter global scope. Node modules 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".

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 10.

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").

To create a 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 visa 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 we `require` the "currency.js" module. The path to our module that we give to the `require` function begins with "./", indicating that the module exists in the same directory as our application script. This means that if you were to create your application script at "/home/mike/projects/node/test-currency.js", then your "currency.js" module file would also need to exist in "/home/mike/projects/node/".

After node has located and evaluated our module, the `require` function returns the contents of "exports" defined in the module. We'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.canadianToUS(30));
```

Using this technique of grouping functionality into module files we can avoid the pitfall of ever-growing application scripts.

2.5.2 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 "node_modules" folder.

In the earlier module example, we required "./currency". If you omit the "./" and simply require "currency" Node will follow a number of rules with which it will search for this module. It will first check to see if the module is a core module, one that was installed with Node. If the module isn't a core module, Node will look for a "node_modules" folder in the applications script's directory. If it doesn't find one it will move to the application script's parent directory. It will keep moving from parent directory to parent directory until it reaches the top-level directory. Finally, if it doesn't find the "node_modules" folder in the top-level directory, then it will check the directories in the environmental variable NODE_PATH.

2.5.3 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.

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 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 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 in to 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.

Following is an example of the use of a callback provided, as an argument, to the function `asyncFunction`. In the example the text "I am the voice of the callback!" is logged to the console after 200 milliseconds.

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

asyncFunction(function() {
  console.log('I am the voice of the callback!');
});
```

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 3.3 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 3.4 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.

The following code defines an echo server that will, when a user connects to it using Telnet, simply echo back whatever gets sent to it. The server is an event emitter with a listener defined for "data" events:

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

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

server.listen(8888);
```

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:

```
cashier.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 3.5 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;                                    ❶
  this.subscriptions[id] = function(senderId, message) {
    if (id != senderId) {                                       ❷
      this.clients[id].write(message);
    }
  }
  this.on('broadcast', this.subscriptions[id]);                 ❸
});

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

❶ **Store reference to this client**
❷ **Ignore data if it's from the receiving client**
❸ **Add send function as broadcast event listener**
❹ **Emit channel join event when client connects to server**
❺ **Emit channel broadcast event when data received by server**

When a user connects to the server❹, a "join" event is emitted to the channel, with the user's remote address/port used as an identifier. The user's connection data is stored❶ and a "broadcast" listener for that user is created❸ that ignores any messages broadcast by the user herself❷. When any user sends data to the server, a "broadcast" event is emitted to the channel❺ specifying the user who created it and the message.

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 8000
```

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 with two new methods.

Listing 3.6 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

One method cycles through the directory 1, processing any files found 2. The other method starts the directory monitoring 3. 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: "serial" and "parallel".

Tasks that need to happen one after the other are called "serial". They are similar conceptually to synchronous logic. As figure 3.1 illustrates, one task completes, then the next, and so on. 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.

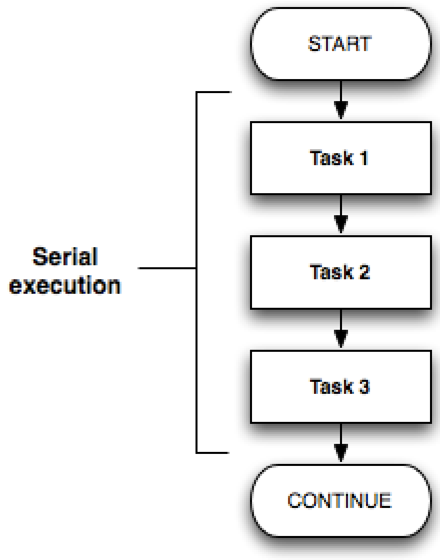


Figure 3.1 Serial execution of asynchronous tasks is similar, conceptually, to synchronous logic: tasks are executed in sequence.

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.

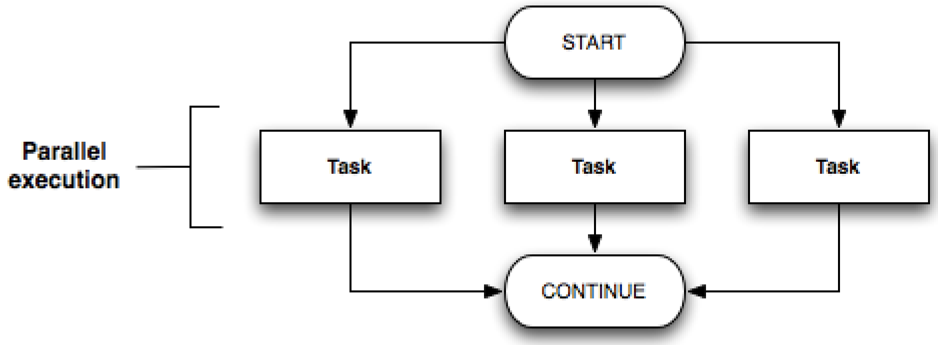


Figure 3.2 Parallel execution of asynchronous tasks allows tasks to execute simultaenously. Only once all tasks are completed, however, will subsequent logic execute.

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 exececssive 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 3.7 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 will act as a queue: when you finish one task you extract the next task in sequence from the array.

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 3.8 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 = [
  function() {
    var configFilename = './rss_feeds.txt';
    path.exists(configFilename, function(exists) {
      if (!exists) {
        next('Please create a list of RSS feeds in $HOME/.rss_feeds.txt.');
      } else {
        next(false, configFilename);
      }
    });
  },
  function (configFilename) {
    fs.readFile(configFilename, function(err, feedList) {
      if (err) {
        next(err.message);
      } else {
        feedList = feedList.toString().replace(/^\s+|\s+$/g, '').split("\n");
        var random = Math.floor(Math.random() * feedList.length);
        next(false, feedList[random]);
      }
    });
  },
  function(feedUrl) {
    request({uri: feedUrl}, function(err, response, body) {
      if (err) {
        next(err.message);
      } else if(response.statusCode == 200) {
        next(false, body);
      } else {
        next('Abnormal request status code.');
      }
    });
  },
  function(rss) {
    var handler = new htmlparser.RssHandler();
    var parser = new htmlparser.Parser(handler);
    parser.parseComplete(rss);

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

function next(err, result) {

  if (err) throw new Error(err);

  var currentTask = tasks.shift();
```

1 Check for file containing list of RSS feeds

2 Read file containing feeds

3 Select random feed from file

4 Do HTTP request for selected feed

5 Parse RSS data into array of items

6 Display title and link of first feed item

7 Handle error, if any

8 Get next task from

```
  if (currentTask) {                                          queue
    currentTask(result);                                      ❾ Execute task
  }
}

next();
```

The application's core functionality is composed of a number of tasks: checking for the existance of a configuration file❶, reading the configuration file❷ and selecting a random RSS feed URL❸, performing an HTTP request for the selected RSS feed❹, and parsing the feed data❺ to display the first item's title and link URL❻. The `next` handler is called if a task runs into an error or properly completes. If an error was reported, the application halts with a stack trace❼. If not, the next task, if any, is pulled from the queue❽ and executed❾. If no more tasks exist, the application is complete.

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. 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 3.9 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) {                    ❶ Output word
      console.log(index +': ' + wordCounts[index]);     counts when
    }                                                    complete
  }
}

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

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

❷ **Count word occurrances in text**

❸ **Get list of files**

❹ **Define task to handle a single file**

❺ **Add task to array**

❻ **Execute tasks in parallel**

The script first gets a list of the files in the "text" directory❸. A task is created ❹, for each file in the list, that includes a call to a function that will count❷ the file's word usage. Each task is added to an array❺. These tasks will be executed❻ using parallel control flow. Finally, when all tasks have completed, the script displays a list of each word used in the files and how many times it was used❶.

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 3.10 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) {    #1
  var url = 'http://nodejs.org/dist/node-v' + version + '.tar.gz';
  var filepath = destination + '/' + version + '.tgz';
  exec('wget ' + url + ' -O ' + filepath, callback);
}

flow.series([                                                      #2
  function (callback) {
    flow.parallel([                                                #3
      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(                                                          #4
      'tar cvf node_distros.tar /tmp/0.4.6.tgz /tmp/0.4.7.tgz',
      function(error, stdout, stderr) {
        console.log('All done!');
        callback();
      }
    );
  }
]);
```

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

The script defines a helper function❶ that will download any specified release version of the Node source code. Two tasks are then executed in series❷: the parallel downloading of two versions of Node❸ and the bundling of the downloaded versions into a new archive file❹.

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
- Importance of server error handling
- Building a RESTful web service
- Serving static files
- Accepting user input from forms

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, we 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. In this chapter we will become familiar with the tools Node provides us to create HTTP servers, as well as get our hands on the filesystem module, necessary for serving static files.

We will also touch on handling additional common web application needs such as HTTP Cookies to provide state, how error handling with Node differs from many frameworks, creating an example low-level RESTful web service, form user input and file upload progress, and finally securing our web application with Node's secure socket layer.

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'll find in familiar frameworks or languages such as PHP, with the ultimate goal of being fast, and flexible.

To begin our mission of creating a robust and performant web application we'll take a look at how Node presents incoming HTTP requests to developers, how you can deal with request related errors, and how to implement some higher level concepts on Node's fundamental building blocks such as reading and writing HTTP cookies.

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

4.1.1 Creating an HTTP server

To begin, you first require the `http` module, and invoke the `createServer()` method with a single argument, a callback. This callback accepts the `request` and `response` objects, which are commonly shortened to `req` and `res`. This allows Node to accept incoming connections to begin parsing requests then you, the developer can apply application logic.

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

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

Once a connection is established, and a request has been made, Node's powerful http-parser comes into play. Node needs only to parse the header before both the `IncomingMessage` and `ServerResponse` instances are provided to the request handler. This greatly differs from the approach of PHP, where the entire program is executed in context of a request. Node servers are long-running processes which will serve many requests throughout it's life-time.

Now let's get on with the fun. To implement the famous "hello world" example, invoke the aptly named `res.write()` method on the response object, writing data to the socket. Then use the `res.end()` method is used to finish the response.

```
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's socket to an address, and listen for connections. The `server` object returned by `http.createServer()` is an instance of `http.Server`, inheriting from `tcp.Server` defined in Node's `net` module. The method that is used to listen for connections, is named `server.listen()`, and is defined within this `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 a unprivileged port such as `3000` on the loopback interface (127.0.0.1), for local use.

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

Sometimes it's useful to have the operating system assign a port for you, these are called ephemeral (short-lived) ports. An example use-case is for testing your web application, as you may want to spawn several instances of it for parallel testing, however you don't want to manage a range of ports manually. To do this, pass `0` to `server.listen()`, after which you access the assigned port by invoking the `server.address()` method which is essentially the `getsockname()` syscall.

```
server.listen(0);
console.log(server.address());
// => { address: '0.0.0.0', port: 53313 }
```

Now that you are set up for accepting connections and handling requests, you may visit "http://localhost:3000" in your browser, resulting in a page consisting of "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 introducing higher level concepts using the APIs Node provides. Next we'll take a look at responding to incoming requests.

4.1.2 Responding to requests

Node v0.4.0 introduced several methods relating to progressive HTTP header field management. Previously only the `res.writeHead(status, fields)` method was available, meaning you had to build up an object containing the fields, which in turn was not very framework friendly. As of Node 0.4.x we have the `res.setHeader(field, value)`, `res.getHeader(field)`, and `res.removeHeader(field)` methods, maintaining the object behind the scenes.

Below is an example of using this progressive API, setting a few header fields, and writing "Hello World" to the socket. When the `res.end()` method is called, Node checks if the header has been written, if not, it writes the header before the remaining data.

```
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

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

This new API also allows setting the response status code in a progressive manner using the `res.statusCode` property, defaulting to 200 "OK". For example you may want to redirect the client with 302 "Found" to indicate that the resource lives at a different location. 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()`, which will flush the header. 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);
```

The older non-progressive API may still be used, however both the status code and the fields must be specified in a single method call.

```
var url = 'http://google.com'
  , body = '<p>Redirecting to <a href="' + url + '">'
    + url + '</a></p>';

res.writeHead(302, {
    'Location': url
  , 'Content-Length': body.length
  , 'Content-Type': 'text/html'
});

res.end(body);
```

Now that you are armed with the basic building blocks of an HTTP server, we will take a look at how to handle errors and why conventional JavaScript error handling doesn't apply to an event loop, however it's critical that a web application is stable.

4.1.3 Error handling

Error handling is an extremely important aspect of designing web applications, and this no exception with Node. In this section we will take a look at how Node web applications handle errors.

Typically with synchronous I/O or any other synchronous task, you may wrap the chunk of code in a `try/catch` statement. When an exception occurs the stack unwinds, passing it to the `catch` block. While this is still true in JavaScript as a language, it does not apply to asynchronous tasks. At a glance the code in listing 4.1 appears to be correct, and respond with "an error occurred!", however it will not work as you may be used to.

Listing 4.1 http_server.js Example HTTP server with incorrect error handling

```
var http = require('http');
var server = http.createServer(function(req, res){
  try {
    query(function(err, result){
       if (err) throw err;
       res.end(result);
    });
  } catch (err) {
    res.end('an error occurred!');
  }
});
server.listen(3000);
```

Throwing exceptions within callbacks is a fundamental change for those used to synchronous idioms, a very important one. The following output shows the default behaviour of Node on an uncaught exception, which is to write the stack trace to stderr, which as you can see, does not contain any of the functions in our server. This is because the event loop loses context of the original stack, when the callback is invoked with our exception (or database results), these call sites no longer represent what we may expect visually.

```
node.js:134
        throw e; // process.nextTick error, or 'error' event
        ➥on first tick
        ^
Error: query failed!
    at Array.<anonymous>
    ➥(/Users/tj/Projects/node-in-action/source/error-handling.js:4:8)
    at EventEmitter._tickCallback (node.js:126:26)
```

The convention of passing the error to callback functions forces Node developers to put more thought into error handling, as this default behaviour can take an entire process down. `try/catch` statements up the stack will not help us catch async exceptions.

Consider the follow synchronous example, where the call stack contains references to the `foo` and `bar` functions, as well as the `try / catch` statement.

Listing 4.2 synchro_error.js Correct synchronous error handling with try catch

```
function bar() {
  throw new Error('oh no!');
}
function foo() {
  bar();
}
try {
  foo();
} catch (err) {
  // handle error
}
baz();
```

The following diagram (figure 4.1) illustrates our previous example. Execution continues "downwards" until an exception is thrown, the call stack then unwinds, that is v8 traverses the call stack in reverse order until a `try / catch` statement is found, and control is passed to the `catch` block. V8 then continues on with execution invoking the final `baz` function.

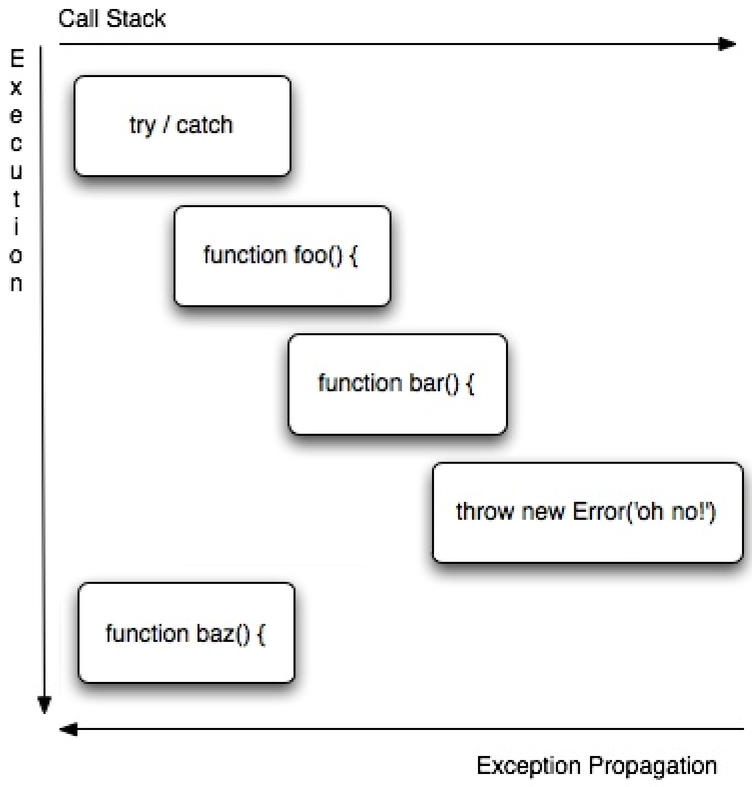


Figure 4.1 Call stack of a synchronous task

Now consider the asynchronous example shown in listing 4.3, identical to the previous, with the exception that the `bar` function invocation is deferred until the "next tick". The `process.nextTick()` function enqueues the callback to be executed when the event loop is idle, essentially making the `foo` function async, because `bar` is not invoked immediately.

Listing 4.3 incorrect_asynch.js Incorrect asynchronous error handling with try catch

```
function bar() {
  throw new Error('oh no!');
}
function foo() {
  process.nextTick(bar);
}
try {
  foo();
} catch (err) {
  // handle error
}
baz();
```

The following diagram (figure 4.2) is similar to the synchronous version (figure 4.1), however you'll see that the execution no longer follows the same flow. The `foo` function is called as before, however since `bar` is now deferred no exception is thrown at this point, and V8 continues on by executing `baz`. Later on when Node's event loop is idle, the enqueued callbacks given to `process.nextTick()` are executed, in this case only the `bar` function, which then throws the exception "oh no!". Following the same rules of exception propagation as the synchronous example, the `foo` function and `try / catch` statement are no longer part of the call stack, only Node internals, forcing it to take action, which we will discuss next.

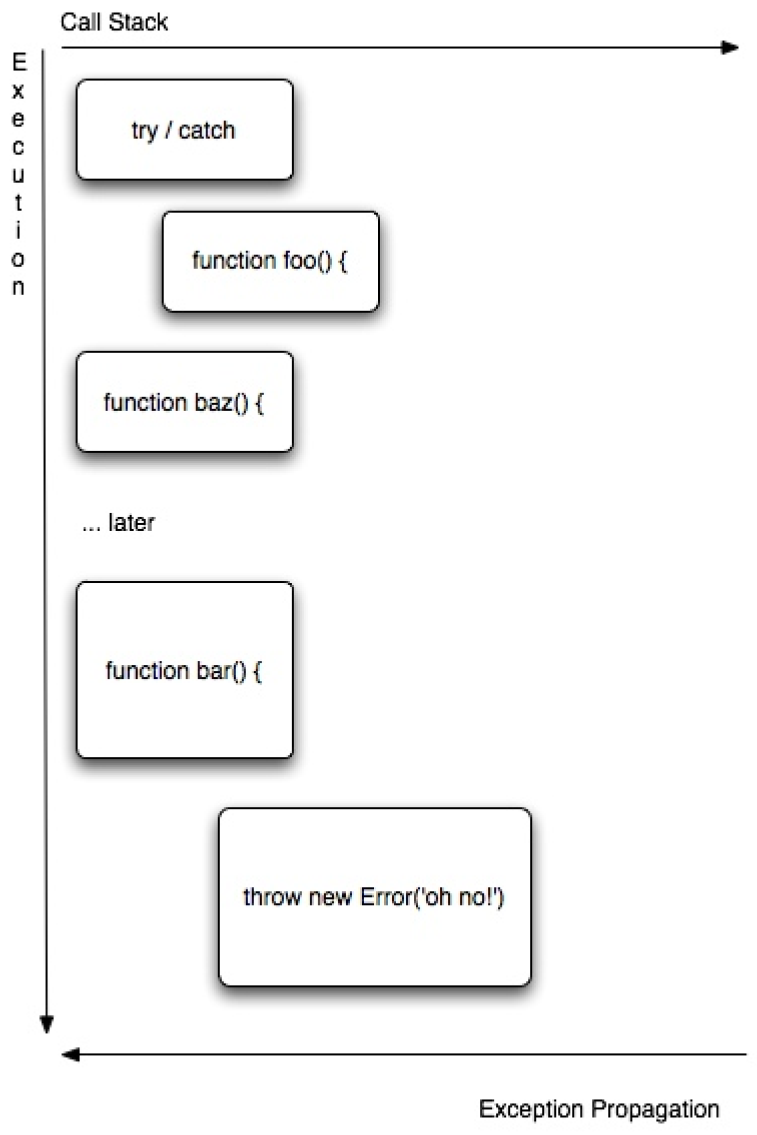


Figure 4.2 Call stack of a synchronous task

So what is a Node developer to do? One way Node helps us with this situation, is by providing the `uncaughtException` event, which is emitted when an exception bubbles to the global scope. When `uncaughtException` has listeners, Node will not exit the process, allowing us to perform customized error reporting such as sending an email notification, or sending an HTTP request to a remote logging system. The `console.error()` method writes to stderr synchronously, thus is more reliable than `console.log()`. Writing to stderr also allows us to separate general logging and error logging.

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

You may have noticed `uncaughtException` doesn't give us access to the request or the response, meaning we cannot respond at this point. By default Node will timeout the socket after two minutes of inactivity, after which it closes the connection. Depending on your application this may not be desirable, as it consumes additional resources, and the client may be unresponsive for those two minutes, this is almost always unacceptable.

You might think simply registering an `uncaughtException` listener and ignoring the error might be a good idea, as it would keep our server alive. However depending on the logic employed by the application, this may leave the application in an unpredictable state causing further issues later on. Typically it is recommended to exit the process, and utilize a process manager such as Cluster to re-spawn the "worker" process, which is discussed in the Deployment chapter.

So how do we combat this issue? Using the previous HTTP server example, we can fix our error handling by immediately responding in the callback, removing the `try/catch` statement.

```
var http = require('http');
var server = http.createServer(function(req, res){
  query(function(err, result){
    if (err) return res.end('an error occurred!');
    res.end(result);
  });
});
server.listen(3000);
```

The `if (err) return ...` statement may appear weird at first, however it's functionally equivalent to the following code, and is a common pattern in node modules, as return values in callbacks generally have no meaning, but help keep the code concise.

```
if (err) {
  res.end('an error occurred!);
} else {
  res.end(result);
}
```

Higher level web frameworks help ease this mundane task, and unify error handling to a single point in the application instead of responding to each case individually. Node's philosophy is to provide a 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. Since nearly all web applications require some form of state, let's take a look at how you could implement this without the support of third-party libraries.

4.1.4 Stateful HTTP with cookies

HTTP "cookies" help provide state to an otherwise stateless protocol. Typically the user's interaction with a web application is limited to a sequence of requests and responses, with no state related. The classic example of this is implementing a transient shopping cart, containing pending purchases.

As mentioned Node's core does not parse the "Cookie" request header field for us, nor does it serialize outgoing "Set-Cookie" response header field, both crucial to working with cookies throughout the request / response cycle. So let's discuss how this can be performed without leveraging a higher level framework, to get a better understanding for working with Node's request and response objects.

PARSING COOKIES

As with other header fields, Node does not alter their contents, so the property `req.headers.cookie` will simply contain the "raw" string when present. To turn this string into something useful within our application we need to parse it, the grammar for this is defined as follows:

```
av-pairs        =       av-pair *(";" av-pair)
av-pair         =       attr ["=" value]
attr            =       token
value           =       word
word            =       token | quoted-string
```

The "Cookie" value of "sid=dG9iaSBydWxlcyEK; name=tobi" would ideally be provided to our application as a JavaScript object containing keys and associated values such as `{ sid: 'dG9iaSBydWxlcyEK', name: 'tobi' }`. Some languages and frameworks do this work for you, a great example is PHP's `$_COOKIE` associative array, providing this feature at the language level. The implementation of the cookie parsing function `parse()` first passes the argument `cookie` to `String()`, which is equivalent to `new String()`, preventing `undefined` values from breaking when the string method `.split()` is invoked, as `null`, `undefined` and other non-strings do not have this method. Next we need to split on the ";" delimiter, allowing leading / trailing white-space, so that we may operate on pairs directly.

```
function parse(cookie) {
  return String(cookie).split(/ *; */).reduce(function(obj, pair){
    return obj;
  }, {});
}
```

Next we use the string `indexOf()` method to return the index of the "=" character, allowing us to slice the key and value apart. We then `trim()` these values to prevent undesired leading or trailing white-space from being included.

```
function parse(cookie) {
  return String(cookie).split(/ *; */).reduce(function(obj, pair){
    var i = pair.indexOf('=')
      , key = pair.slice(0, i).trim().toLowerCase()
      , val = pair.slice(i + 1, i.length).trim();
    if ('"' == val[0]) val = val.slice(1, -1);
    obj[key] = val;
    return obj;
  }, {});
}
```

The implementation of our parser will now parse the following Cookie strings and provide an object as expected.

```
console.log(parse('sid=dG9iaSBydWxlcyEK; name=tobi'));
console.log(parse('SID   = dG9iaSBydWxlcyEK; name  = "tobi"'));
// => { sid: 'dG9iaSBydWxlcyEK', name: 'tobi' }
// => { sid: 'dG9iaSBydWxlcyEK', name: 'tobi' }
```

Now that we can access incoming cookie information, we need a way to set them in the first place! Let's take a look at serializing a cookie object into a string.

SERIALIZING COOKIES

Serializing a cookie is essentially the same process as parsing it, in reverse, however when instructing the browser to store a cookie we can provide additional key/value pairs such as "path", "expires", "httpOnly" and others, instructing the browser of how to store and expose this data.

When responding to a request the "Set-Cookie" header field may be set several times to set many cookies, the simplest form of this may look like this:

```
Set-Cookie: name=tj
```

By default "path" and "domain" scope to the current pathname and domain of the original request. This means the cookie "name=tj" for a request to "example.com/admin" will be available only to requests starting with the pathname "/admin". It's important to note, as authenticating a user on a request such as "/login" will then be scoped to "/login", which is often an undesired behaviour.

Serializing a cookie value is pretty straight-forward. The `serialize()` function expects the cookie `name`, `val`, followed by an optional object representing the attributes. We push each pair to an array, then join with the ";" delimiter upon return.

```
function serialize(name, val, obj) {
  var pairs = [name + '=' + encodeURIComponent(val)]
    , obj = obj || {};
  if (obj.domain) pairs.push('domain=' + obj.domain);
  if (obj.path) pairs.push('path=' + obj.path);
  if (obj.expires) pairs.push('expires=' + obj.expires.toUTCString());
  if (obj.httpOnly) pairs.push('httpOnly');
  if (obj.secure) pairs.push('secure');
  return pairs.join('; ');
}
```

Give it a try! The following calls should output valid "Set-Cookie" values.

```
console.log(serialize('name', 'tj'));
console.log(serialize('name', 'tj', { path: '/', httpOnly: true }));
// => "name=tj"
// => "name=tj; path=/; httpOnly"
```

With this complete, setting a cookie is as simple as setting the fields:

```
res.setHeader('Set-Cookie', serialize('name', 'tj'));
res.setHeader('Set-Cookie', serialize('name', 'tj', { path: '/' }));
```

Receiving and setting cookies paves the way for more complex solutions such as implementing sessions, another important feature that Node itself does not implement, but don't worry! The third-party framework "Connect" implements a robust session implementation. For now let's retain our focus on interacting with Node at a lower level as it's crucial for framework implementors, and for a better understanding of Node as a platform. To continue let's build a RESTful web service, and highlight some of Node's utility modules such as "url" for parsing urls, and "querystring" for parsing query-strings, both essential for routing and handling requests appropriately.

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 authors 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 browser to interact with our web service. `curl` is a powerful command-line HTTP client which can be used to send requests to a target server. First we'll prepare the application to add items via POST, followed by a listing with the GET verb, finishing with item removal with DELETE.

4.2.1 Creating resources with POST requests

To get started the todo list needs to expose a means to create list items. For the data store a regular JavaScript array will be used to replace a database as shown in the following code:

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

var items = [];

var server = http.createServer(function(req, res){

});

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

As mentioned the creation of a resource is typically mapped to the POST verb, which can be accessed via the `req.method` property as shown in listing 4.4, giving us an idea of which task to perform.

As Node's http-parser streams data we receive "chunks" of data in the form of

`Buffer` objects, or strings depending on the encoding. It is up to you to decide if you must collect or "buffer" all of this data in order to work with it, or if it can be processed in streaming manner.

By default the "data" events provide `Buffer` objects, which are essentially byte arrays, as neither JavaScript nor Google's V8 provide a performant or convenient structure for interacting with arbitrary binary data. In the case of textual todo items, there's little interest in binary data, so setting the stream encoding to "ascii" or "utf-8" is ideal, as it will emit strings. This can be done by either invoking `req.setEncoding(encoding)` which applies to all data chunks, or calling `chunk.toString(encoding)` on each `Buffer` object.

In the case of a todo list item, the entire string is needed before it can be added to the array. One way to do this is concatenate all of the chunks until the "end" event is emitted, indicating the request is complete. At this time you will now have the `item` string populated with the entire contents of the request body as shown in the following listing, which can then be pushed to the `items` array. Now that the item has been added we can end the request with the string "OK" and Node's default status code of 200. The use of `res.end(data)` in listing 4.4 functionally equivalent to a `res.write(str)` call followed by a call to `res.end()`.

Listing 4.4 todo.js POST request body string buffering

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

Now that the application can add items, the next task is listing the items.

4.2.2 Fetching resources with GET requests

To handle the GET verb add it to the same switch statement, followed by the items list logic. The following example shows one way this may be implemented using several calls to the response `write()` method, followed by `end()` to complete the response. The first call to `res.write()` will write the header with some 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 you can give it a try! Fire up a terminal and POST some items using curl's `-d` flag:

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

Then to GET the list 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
```

Using a command-line program like `telnet` or `netcat` you can issue the same request by hand by typing "GET / HTTP/1.1" followed by a return. In listing 4.5 you'll see that Node has set the `Transfer-Encoding` field to "chunked", this is because of the multiple calls to `res.write()` without setting the "Content-Length" field, producing a response body with an unknown length.

When the request is issued with HTTP 1.1 Node sets an internal flag named `useChunkedEncodingByDefault` which will write each chunk with it's associated length as a hexadecimal, followed by `0` indicating the end of the body. Note that Node also sets the `Connection` field to `keep-alive`, indicating that the server wishes to keep the TCP connection established for subsequent requests.

Listing 4.5 Chunked HTTP response

```
$ telnet localhost 3000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GET / HTTP/1.1

HTTP/1.1 200 OK
Connection: keep-alive
Transfer-Encoding: chunked

11
0) buy groceries

16
1) buy node in action

0
```

When the same request is issued with HTTP 1.0, Node will set `Connection` to "close" and close the connection.

```
$ telnet localhost 3000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GET / HTTP/1.1



HTTP/1.1 200 OK
Connection: close



0) buy groceries
1) buy node in action
```

To speed up responses, when possible the "Content-Length" field should be defined, in the case of the item list, the body can easily be constructed ahead of time, allowing us to access the string length.

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

The string length should not be used to populate this field in the case that it contains multi-byte characters, as "Content-Length"'s value should represent the byte-length, not character length, which is value of `body.length`. For this reason 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
```

While this method works and is fine for small responses, this is not always feasible when dynamically generating content. Coming up in the next section we will discuss this further.

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 be aware of the requested url, the HTTP client will specify which to remove, for example `DELETE /1` or `DELETE /5`.

The 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`. Thankfully Node provides a core module for parsing urls, aptly named "url". The following shows how the parser separates these components:

```
require('url').parse('/1')
// => { pathname: '/1', href: '/1' }

require('url').parse('/1?api-key=foobar')
// => { search: '?api-key=foobar',
  query: 'api-key=foobar',
  pathname: '/1',
  href: '/1?api-key=foobar' }
```

As shown in listing 4.6 the path may be accessed directly, 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 specific to this url is to use the `String#slice()` method, used to return a portion of the string between two indices, in this case skipping the first character, as the second argument is optional, defaulting to the length of the string. To convert this string to a number it should be passed to the JavaScript global function `parseInt()` which accepts a radix argument to specify the numeral system base to recognize. In this case a base 10 integer, otherwise `parseInt` will attempt parsing other literals such as a base 16 hexadecimal string "0xff".

If the number is "Not a Number", the status code is set to 400 indicating a "Bad Request". Following that we check if the item even exists, responding with 404 "Not Found", and finally when the item is successfully removed the app responds with 200 "OK".

Listing 4.6 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');
  }
  break;
```

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, but learning the fundamentals of Node is crucial for understanding, debugging, and enables you to create more powerful applications and frameworks. We're not through yet! next we'll take a close look at serving static files with Node's file-system or "fs" module.

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, our implementation in this section illustrates Node's low level filesystem API.

We'll also take a look at benchmarking our static file server using Apache Bench, an small command-line program originally developed for testing the popular Apache web server.

Finally we will cover how you can respond with the appropriate response codes when a file is missing, or when an error occurs, but first let's get started with the server.

4.3.1 Creating a static file server

The core of our file server looks much like any other HTTP server you've read up to this point. In the following snippet you'll see that this time a variable named `root` defined, which will act as the static file server's root directory. With this defined you can restrict access to any files requested "above" root.

```
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){
});

server.listen(3000);
```

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. Node also provides the `__filename` variable which is an absolute path to the script itself.

The next step is accessing the pathname of the url in order to determine the requested file path. To do this Node provides the "url" module, specifically the `.parse()` function exported. The following node REPL session illustrates the use of this function, splitting up the url into an object, including the `pathname` property we're interested in.

```
$ 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: '/' }
```

If the requested path was "/js/jquery.js", and our `root` path was "/var/www/example.com/public" you can simply join these using the "path" module's `.join()` method to form our absolute path "/var/www/example.com/public/js/jquery.js".

```
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 a `fs.ReadStream`, an object inheriting from `Stream.protoype`, which itself inherits from `EventEmitter.prototype`, emitting "data" events as it incrementally reads our file. This implementation is now a fully functional file server, however there many more details we will discuss.

Listing 4.7 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);               ❶ Get absolute path
  var stream = fs.createReadStream(path);            ❷ Create ReadStream
  stream.on('data', function(chunk){                 ❸ Write response
    res.write(chunk);
  });
  stream.on('end', function(){
    res.end();                                       ❹ End the response
  });                                                   when file is
});                                                     complete
server.listen(3000);
```

While it's important to know how the `fs.ReadStream` works, and the flexibility its events provide, Node provides a higher level mechanism 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 us to greatly simply the implementation, and handle TCP back pressure appropriately.

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

To confirm that the static file server is functioning, you can execute 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 it-self, which is the response body of the following request:

```
$ 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
...
```

Now that the server is functioning, you'll probably want to get an idea of how it performs. In the next section we'll take a look at the Apache Bench tool designed to do exactly that.

4.3.2 Benchmarking your server

A common and simple way to get an idea of how your server performs is to use the "Apache Bench" command-line program, or "ab". This program was originally part of the Apache web server, however it is available as a stand-alone download as well. Apache Bench issues the number of requests specified with the "-n" flag, at a concurrency level specified with "-c".

```
$ ab -n 5000 -c 80 http://localhost:3000/static.js
```

Apache Bench will output several statistics, shown in the following listing, these range from the over-all time taken, an average of requests per second, the total amount of data transferred along with others.

Listing 4.8 ApacheBench static file server benchmark output

```
Document Path:          /static.js
Document Length:        643 bytes
Concurrency Level:      80
Time taken for tests:   1.862 seconds
Complete requests:      5000
Failed requests:        0
Write errors:           0
Total transferred:      3510000 bytes
HTML transferred:       3215000 bytes
Requests per second:    2684.57 [#/sec] (mean)
Time per request:       29.800 [ms] (mean)
Time per request:       0.372 [ms]
➥ (mean, across all concurrent requests)
Transfer rate:          1840.40 [Kbytes/sec] received
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.3      0       3
Processing:    17   29   5.8     29      46
Waiting:       11   22   5.9     22      39
Total:         17   30   5.7     29      46
```

Along with the mean time spent per request in milliseconds, `ab` outputs a useful list at the bottom of this report containing the tail-end of the response times, giving you a good idea of how your application may degrade over time.

```
Percentage of the requests served within a certain time (ms)
  50%     29
  66%     32
  75%     33
  80%     34
  90%     38
  95%     40
  98%     42
  99%     43
 100%     46 (longest request)
```

Realistic benchmarking is tricky business, and out of scope for this book, however these quick benchmarks provide a rough idea of how your server will perform. These averages mean very little unless benchmarking before making substantial changes to your application to reference, or perhaps running similar benchmarks with other frameworks on the same machine to see how they compare. The most important aspect of optimization is to profile and determine bottlenecks, which we will be discussing in the chapter "Debugging Node applications".

So far our file server logic contains no error handling, making it very prone to crashing, taking the entire process with it, not just a single request. In the next section we'll provide appropriate error handling making our file server more robust.

4.3.3 Handling server errors

Up until now the static file server does not apply any logic to handle files which are not found, attempted access to forbidden files, or I/O related errors. In this section we'll touch on how you can make your file server, or any node server more robust.

By default node's "error" events will throw when no listeners are present, and this is true for any stream, even the `fs.ReadStream` instance serving a static file. This is important to know, because if left un-handled 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'll need to register an "error" event handler, 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');
});
...
```

Something to keep in mind is that an error may have emitted before the response header was written, or after. If the response header is already written to the client we have no chance to report a nice message like "Failed to transfer your file", because a portion of it may have already been transferred. On the other hand if the response header has not yet been written we can simply alter the response status code and write a useful message.

As of node v0.4.10 the public API does not contain the means to determining if the header has been written or not, however there is a private flag named `res._headerSent`, though a public flag has been proposed and may be included in a future release. In the event that an error occurs and the header has been sent there is little you can do, however at very least you should log the exception:

```
stream.on('error', function(err){
  if (res._headerSent) {
    console.error(err);
  } else {
    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.

The refactored file server shown in listing 4.9 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 4.9 file_server.js File server checking for existance and responding with Content-Length

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

1. **parse out pathname**
2. **get absolute path**
3. **check file existence**
4. **file doesn't exist**
5. **some other error**

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 develop, user input from forms.

4.4 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 aspect of Node, this separation of concerns is powerful, leaving opinions to third-party frameworks and providing a fast, robust networking solution. To get started we'll take a look at handling submitted form fields, handling uploaded files, and reporting upload progress in pseudo real-time.

4.4.1 Form submission

Typically two Content-Types 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 we will re-write our todo list from "Building a RESTful web service" to utilize an html form.

To get started a switch on the request method, or `req.method` is used in listing 4.10 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".

Listing 4.10 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);
```

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');
}
```

Next up is handling the GET request, responding with the following html form:

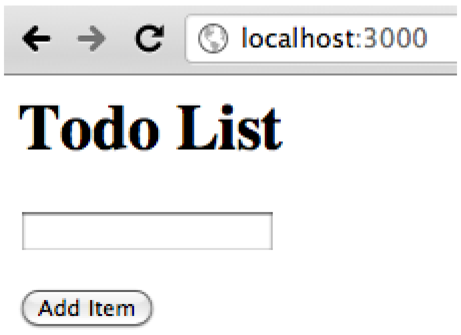


Figure 4.3 A todo-list application created without the support of a web framework

Though typically markup is generated using template engines, for simplicity the following example just uses string concatenation, other than this the function looks very similar to the previous two, however this time there is no need to assign `res.statusCode` because it defaults to 200 "OK".

Listing 4.11 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('\n')
    + '</ul>'
    + '<form method="post" action="/">'
    + '<p><input type="text" name="item" /></p>'
    + '<p><input type="submit" value="Add Item" /></p>'
    + '</form>';
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('Content-Length', html.length);
  res.end(html);
}
```

Finally the application needs to implement the `add()` function which will accept both the request and response 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 it will not dealing with binary data, you may set the request encoding type to "utf8" with `res.setEncoding()`, signalling Node to decode the data before emitting the "data" event, which now provides strings instead of a `Buffer` object, although this is functionally equivalent to `chunk.toString('utf8')`, which may be called on individual `Buffer` instances, even alternating encodings if necessary for the application.

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 a 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, helping to prevent resource starvation. This process incrementally parses the "data" chunks as they are emitted, though this is more difficult to use and implement.

The following example utilizes Node's querystring module to parse the body. If the item "take ferrets to the vet" was submitted the body would contain "item=take+ferrets+to+the+vet", which is passed to `qs.parse()`, returning the JavaScript object `{ item: 'take ferrets to the vet' }`. Following adding of the item, a call to the same `show()` function previously implemented will display the form again, which in the future could potentially display a message such as "Added todo list item". Another approach would be to redirect 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.

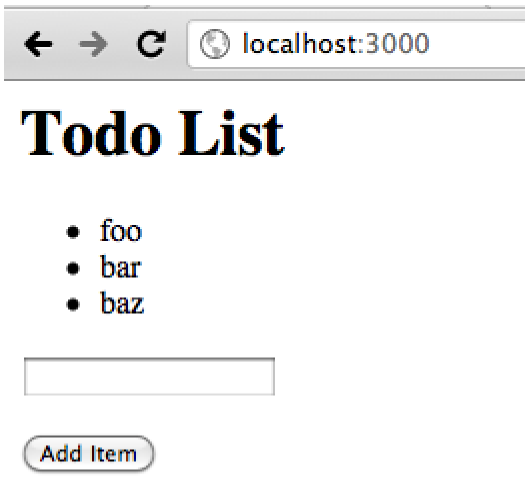


Figure 4.4 The Node todo list application in action

4.4.2 Uploaded Files

Handling of uploads is another very common, and important aspect of web development. Much like handling "application/x-www-form-urlencoded" bodies Node simply emits data in arbitrary chunks, however in the case of file uploads the "multipart/form-data" Content-Type. In this section we will take a look at what is involved in handling uploads, as well as 3rd-party modules lending developers a hand.

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

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

Note that the "enctype" attribute is not present, as the browser will default it to "application/x-www-form-urlencoded", and the request body may look similar to below, containing the file's name rather than the file data.

```
name=my+clock&file=clock.png
```

To handle this properly and accept the file's content the "enctype" or the "encoding type" attribute should be set to "multipart/form-data", a MIME type better suited for large BLOBs (Binary Large Objects), in which each "part" contains it's own header describing its body.

```
<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>
```

Upon submission the browser constructs a "multipart" request, a request body consisting of many "parts", each preceded by a "boundary" delimiter which is a Carriage Return and Line Feed (CRLF), followed by the boundary string itself. The boundary for a multi-part body is defined in the Content-Type header field, automatically generated and populated assigned by the browser as shown in the following field:

```
Content-Type: multipart/form-data; boundary=
➥----WebKitFormBoundaryQpTTHD1BnfA3Bq1p
```

The following request body contains each form part delimited by the boundary defined in the Content-Type header field, where "PNG..." is a truncated BLOB. Each part may have several fields, much like the HTTP request header itself, for example when submitting "clock.png", browser defined the "Content-Type: image/png" header. Following the parts is the closing delimiter which is the original boundary followed by two trailing hyphens "--", indicating that no more parts are present.

```
------WebKitFormBoundaryQpTTHD1BnfA3Bq1p
Content-Disposition: form-data; name="name"

my clock
------WebKitFormBoundaryQpTTHD1BnfA3Bq1p
Content-Disposition: form-data; name="file"; filename="clock.png"
Content-Type: image/png

PNG...
------WebKitFormBoundaryQpTTHD1BnfA3Bq1p--
```

Parsing multi-part 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 fulfil 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 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.

The following HTTP server 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 4.12 http_server_setup.js HTTP server setup prepared to accept file uploads

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

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

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

function upload(req, res) {
  // upload logic
}
```

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 `handle()` 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;
  }
}
```

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()`.

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

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 following listing, formidable gives you access to events like "field", and "file", along with common events like "end".

Listing 4.13 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
```

While 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.

Listing 4.14 Formidable "file" event output

```
file
{ size: 28638,
  path: '/tmp/d870ede4d01507a68427a3364204cdf3',
  name: 'clock.png',
  type: 'image/png',
  lastModifiedDate: Sun, 05 Jun 2011 02:32:10 GMT,
  _writeStream:
   { path: '/tmp/d870ede4d01507a68427a3364204cdf3',
     fd: 9,
     writable: false,
     flags: 'w',
     encoding: 'binary',
     mode: 438,
     busy: false,
     _queue: [],
     drainable: true },
  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 reporting upload progress, a process which comes quite natural to Node and its event loop.

4.4.3 Upload progress reporting

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 = bytesReceived / bytesExpected * 100;
  console.log(percent);
});
```

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

```
1.320772095276523
2.6633412731528447
4.005910451029166
5.348479628905488
6.691048806781811
8.03361798465813
...
99.32832208024801
100
```

Though this implementation works fine, decimals may not be ideal for the application, and often provide too much information. A simple and efficient trick for stripping is the use of JavaScript's bit-wise operators, which convert the operands to 32-bit signed integers, so you may use a "noop" (having no effect) bit-wise operation such as `n | 0`, or `n >> 0`, to force the number to an integer.

```
form.on('progress', function(bytesReceived, bytesExpected){
  var percent = bytesReceived / bytesExpected * 100 | 0;
  console.log(percent);
});
```

The script now outputs whole numbers:

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

To update a single line with the percentage instead of a sequence of lines, you could change `console.log()` call which appends a newline character, to a `process.stdout.write()` call which provides more direct access. If you write a carriage return followed by the percentage, the terminal will move to the beginning of the line before writing, producing much more pleasant output.

```
process.stdout.write('\r' + percent + '%');
```

Now that we understand these concepts, we can apply them to the browser, a fantastic feature for any application expecting large uploads.

REPORTING PROGRESS TO THE BROWSER

Now that the upload progress is displaying in the terminal, let's take a look at reporting progress to a browser, producing the following result:

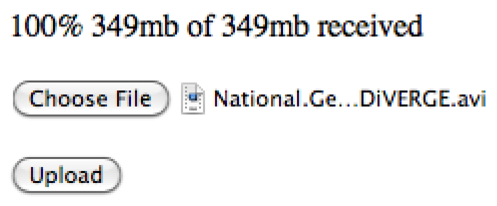

100% 349mb of 349mb received

Figure 4.5 Upload progress indication with no assistance from third-party libraries

This process is fundamentally similar to the previous terminal progress indication, however the browser needs the means to receive this data, as well as map the progress to the original request. Since HTTP is a stateless protocol we, the developers must come up with techniques to provide the state necessary. Modern browsers may achieve this with WebSockets, however older browsers will need fallbacks such as polling with XMLHttpRequests. Implementing a efficient, cross-browser file uploader is a non-trivial task, however we will touch on the basics.

As per usual you'll need an HTTP server, however this time matching the request's url with the regular expression `/^\/upload\/progress\/([^\/?]+)/` as shown in the following listing. This "route" will be used to fetch progress for the upload identified by the capture group's match, stored in `RegExp.$1`, the string matched by the pattern expressed within parenthesis, where a second group would be `RegExp.$2` and so on.

Listing 4.15 upload_progress.js Upload progress route handling

```
var http = require('http')
  , qs = require('querystring')
  , formidable = require('formidable');
var server = http.createServer(function(req, res){
  switch (req.method) {
    case 'GET':
      if (/^\/upload\/progress\/([^\/?]+)/.exec(req.url)) {
        uploadProgress(req, res, RegExp.$1);
      } else {
        show(req, res);
      }
      break;
    case 'POST':
      upload(req, res);
      break;
  }
});
```

The form POST in listing 4.16 handler looks nearly identical to what we've discussed in "Handling Uploads" however this time hidden form input named "id" is necessary to uniquely identify the upload's progress. With this id we can expose the progress data in such a way that it can be accessed for subsequent requests. In case of this example let's store the data mapped by the "id" on a `uploadProgress` object, however for durability and scalability a key/value store such as Redis would be ideal for a production implementation.

Listing 4.16 upload_post.js File upload POST request handler

```
function upload(req, res) {
  if (!isFormData(req)) {
    res.statusCode = 400;
    res.end('Bad Request');
    return;
  }
  var form = new formidable.IncomingForm
    , id;
  form.on('field', function(name, value){
    if ('id' == name) id = value;
  });
  form.on('progress', function(bytesReceived, bytesExpected){
    if (!id) return;
    var percent = bytesReceived / bytesExpected * 100 | 0;
    uploadProgress[id] = {
        percent: percent
      , bytesReceived: bytesReceived
      , bytesExpected: bytesExpected
    };
  });
  form.parse(req);
}
function isFormData(req) {
  return 0 == req.headers['content-type']
  ➥.indexOf('multipart/form-data');
}
```

When the request url matches "GET /upload/progress/<id>", the request, response, and upload id are passed to the `uploadProgress` function shown in listing 4.17, which happens to also be the object used to map these values, as functions may contain properties just like any other object. With this the `uploadProgress[id]` expression can be used to access an upload's progress.

When the `progress` object exists, passing it to `JSON.stringify()` will return a JSON string representation which may be parsed on the client-side to obtain an object, becoming the body of the "application/json" response.

Listing 4.17 upload_json.js JSON upload progress request handler

```
function uploadProgress(req, res, id) {
  var progress = uploadProgress[id]
    , json;
  if (progress) {
    json = JSON.stringify(progress);
    res.setHeader('Content-Type', 'application/json');
    res.setHeader('Content-Length', json.length);
    res.end(json);
  } else {
    res.statusCode = 404;
    res.end('failed to find upload "' + id + '"');
  }
}
```

Lastly the server needs a user interface. Typically a template engine is used for this, however for simplicity the `show()` handler function will concatenate a string as we've done previously. The following form in listing 4.18 is populated with a hidden field named "id" assigned a pseudo-random identifier generated from a timestamp of the current date and `Math.random()`. An iframe with the same id is then created, used as the target for the form so that it will not disrupt the user experience, as the main window would otherwise be the target.

Listing 4.18 upload_progress_page.js HTML page for upload progress reporting

```
function show(req, res) {
  var id = Date.now() + Math.random();
  var html = ''
    + '<html>'
    + '<head>'
    + '<script>' + client
      + '; addEventListener("DOMContentLoaded", function(){'
      + 'client(' + id + ');'
      + '}, false);'
    + '</script>'
    + '</head>'
    + '<body>'
    + '<span id="progress"></span>'
    + '<iframe id="' + id + '" style="display: none"></iframe>'
    + '<form method="post" target="' + id
    + '" action="/" enctype="multipart/form-data">'
    + '<p><input type="hidden" name="id" value="' + id + '" /></p>'
    + '<p><input type="file" name="file" /></p>'
    + '<p><input type="submit" value="Upload" /></p>'
    + '</form>';
    + '</body>';
    + '</html>';
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('Content-Length', html.length);
  res.end(html);
}
```

In the previous listing, you will see that the `client` variable is concatenated right after the "<script>" tag. This is a function defined in listing 4.19, which when concatenated will invoke `Function#toString()`, as string representation of the function itself, which serves as an extremely simple way to expose JavaScript to the client, while maintaining it in a server-side script.

For the purposes of this book the application will implement an iframe transport only, however flash and other existing alternatives are available to produce a robust cross-browser solution. The `client()` function following is invoked on

"DOMContentLoaded" with the generated `id`. Upon form submission it begins polling for upload progress using an XMLHttpRequest, every 300 milliseconds until the percentage has reached 100.

Listing 4.19 upload_client_logic.js Client-side upload progress logic

```
function client(id) {
  var form = document.getElementsByTagName('form')[0]
    , progress = document.getElementById('progress');

  form.addEventListener('submit', poll, false);

  function request(url, fn) {
    var req = new XMLHttpRequest;
    req.open('GET', url, true);
    req.send(null);
    req.onreadystatechange = function(){
      if (4 == req.readyState) fn(req);
    };
  }

  function mb(n) {
    return n / 1024 / 1024 | 0;
  }

  function poll() {
    request('/upload/progress/' + id, function(req){
      var obj, delay = 300;
      if (200 != req.status) return setTimeout(poll, delay);
      res = JSON.parse(req.responseText);
      progress.textContent = res.percent + '%'
        + ' ' + mb(res.bytesReceived) + 'mb'
        + ' of ' + mb(res.bytesExpected) + 'mb received';
      if (100 != res.percent) setTimeout(poll, delay);
    });
  }
}
```

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 Secure HTTP

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.

To generate a private key you'll need OpenSSL, which will 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 the you've generated your keys, put them in a safe place. In our example HTTPS server 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 4.20 https_server.js HTTPS server options

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

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

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

Once the above 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). If you're deploying a public site it's worth it to use a CA to insure secure communications. Registering with a CA is out of the scope of this book, but vendors such as Verisign, usually provide instructions.

4.6 Summary

In this chapter, we've introduced the fundamentals of Node's HTTP server, teaching you how to response to incoming requests, how to handle asynchronous exceptions in order to keep your application reliable, and how to introduce high-level concepts like HTTP cookies, all with nothing more than code Node APIs. You've learned how to create a RESTful web application, serve static files, and even create a pseudo real-time upload progress reporter.

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, Postgres, 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

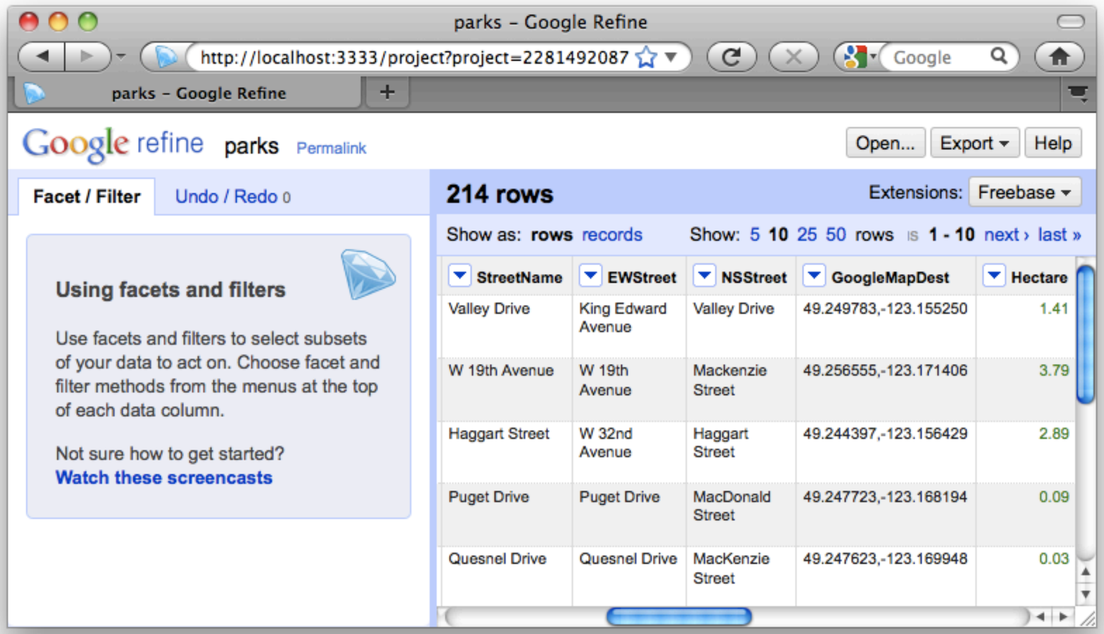
The most convenient storage mechanisms, from the standpoint of systems administration, are mechanisms that don't require you to maintain a DBMS to use them: in-memory storage and file-based storage.

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/


parks – Google Refine
http://localhost:3333/project?project=2281492087
Google
parks – Google Refine
Google refine parks Permalink
Open... Export Help
Facet / Filter Undo / Redo 0
214 rows
Extensions: Freebase
Show as: rows records Show: 5 10 25 50 rows 1 - 10 next › last »

Using facets and filters

Use facets and filters to select subsets of your data to act on. Choose facet and filter methods from the menus at the top of each data column.

Not sure how to get started?
Watch these screencasts

StreetName	EWStreet	NSStreet	GoogleMapDest	Hectare
Valley Drive	King Edward Avenue	Valley Drive	49.249783,-123.155250	1.41
W 19th Avenue	W 19th Avenue	Mackenzie Street	49.256555,-123.171406	3.79
Haggart Street	W 32nd Avenue	Haggart Street	49.244397,-123.156429	2.89
Puget Drive	Puget Drive	MacDonald Street	49.247723,-123.168194	0.09
Quesnel Drive	Quesnel Drive	MacKenzie Street	49.247623,-123.169948	0.03

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

❶ **Make sure todo file exists**

❷ **Read todo data from file**

❸ **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. 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 freelance web development work. You'd need to keep track of, at the least, clients and time billed. You could build this web application using the filesystem as a simple data store, but it would be tricky to build reports. If you wanted to create an invoice for a client, for example, you'd have to read every billing item stored and check to see if it applied to the client and hadn't already been included in a previous invoice.

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

Install `node-sqlite` using the following command.

```
npm install sqlite
```

Listing 5.3 show how to establish a connection to an SQLite database (in this example, accessing data in a file named "tasks.sqlite").

Listing 5.3 sqlite_connect.js Connecting to an SQLite database

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

db.open('tasks.sqlite', function(err) {
  if (err) throw err;
  console.log('We are now able to perform queries.');
});
```

❶ **Put SQLite query code here**

node-sqlite's execute function performs queries of all types. The following example shows the creation of a table.

```
db.execute(
  "CREATE TABLE tasks ("
  + "id INTEGER PRIMARY KEY, "
  + " description TEXT)",
  function(err, rows) {
    if (err) throw err;
    console.log('Table created.');
  }
);
```

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. The following example shows the insertion of a row using this technique. Note that the second argument of the `execute` method is now a list of values to substitute with the placeholders.

```
db.execute(
  "INSERT INTO tasks (description) VALUES (?)",
  ['Take out the trash.'],
  function(err, rows) {
    if (err) throw err;
    console.log('Task added.');
  }
);
```

When issuing a query that will return data, use a callback to handle the data received. The following example shows this in action: outputing the "description" column of each row returned by a query.

```
db.execute(
  "SELECT * FROM tasks",
  function(err, rows) {
    if (err) throw err;
    for(var i in rows) {
      console.log(rows[i].description);
    }
  }
);
```

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 PostGres. MySQL and PostGres 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 PostGres.

5.2.1 MySQL

MySQL is the world's most popular SQL database and is well-supported by the Node community. If you're new to MySQL and interested in learning it, 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
```

Once you've installed the node-mysql module, you can connect to MySQL, and select a database to query, using the following code:

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

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

client.useDatabase('miketest');
```

With a MySQL connection opened, your application will not exit until you close the MySQL connection. If you wanted to end connection from the previous example use the following code.

```
client.end();
```

The `query` method is for all queries in mysql-node. The following example shows how to insert a row into a database table.

```
client.query(
  'INSERT INTO users ' +
  "(name) VALUES ('Mike')"
);
```

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. The following example shows the insertion of a row using placeholders.

```
client.query(
  "INSERT INTO users " +
  "(name, age) VALUES (?, ?)",
  ['Mike', 39]
);
```

Sometimes when you insert a row you need to subsequently know the value of the row's primary key. Do this by adding a callback argument to the query call.

```
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 similarly be a callback. Listing 5.4 shows the use of a callback to output data returned from a query.

Listing 5.4 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();
  }
);
```

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

5.2.2 PostGres

PostGres is well-regarded for its standards compliance and robustness and is the favored RDBM of many Node developers. If you're new to PostGres 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 PostGres 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 PostGres, and select a database to query, using the following code (omitting the ":mypassword" portion of the connection string if no password is set).

```
var pg = require('pg');
var conString = "tcp://myuser:mypassword@localhost:5432/mydatabase";

var client = new pg.Client(conString);
client.connect();
```

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. This object emits a 'row' event for each retrieved database row. Listing 5.5 shows how you can output data from each row returned by a query.

Listing 5.5 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) {          ❶ Handle return of a row
  console.log(row.name)
});

query.on('end', function() {             ❷ Handle query completion
  client.end();
});
```

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);
});
```

Listing 5.6 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 5.6 redis_keys.js Storing data in elements of a Redis hash.

```
client.hset('camping', 'shelter', '2-person tent', redis.print  ❶ Set hash elements
client.hset('camping', 'cooking', 'campstove', redis.print);
client.hget('camping', 'cooking', function(err, value) {      ❷ Get "cooking"
  console.log('Will be cooking with: ' + value);                  element's values
});

client.hkeys('camping', function(err, keys) {                  ❸ Get hash keys
  if (err) throw err;
  keys.forEach(function(key, i) {
    console.log('  ' + key);
  });
});
```

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.

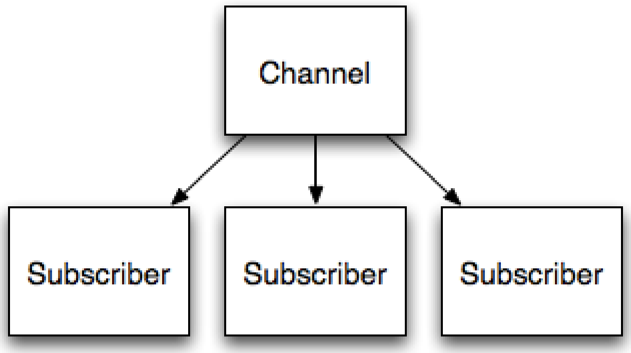


Figure 5.2 Redis channels provide an easy solution to a frequent data delivery scenario.

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.

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.

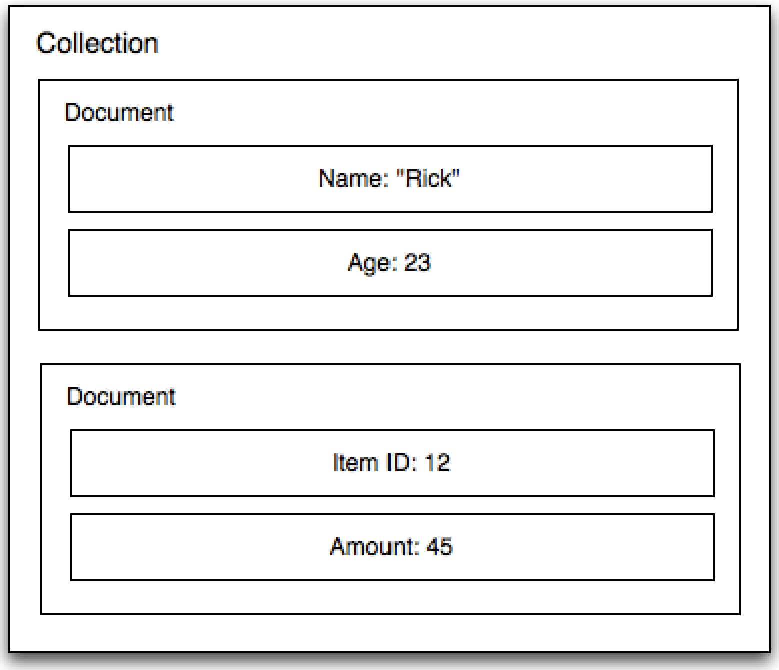


Figure 5.3 If you use MongoDB to store data for your Node application, then each item in your MongoDB collection can have a completely different schema.

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

Listing 5.7 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.');      ❶ 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 5.8 shows how to update a document using its ID.

Listing 5.8 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);
```

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 5.9 shows how you can update a document using Mongoose.

Listing 5.9 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.');
  }
);
```

❶ Update using internal ID
❷ 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're now ready to move on to Connect: a powerful tool for abstracting away much of the difficulty of creating web applications.