

Node.js

IN ACTION

Mike Cantelon
TJ Holowaychuk
Nathan Rajlich



 MANNING



MEAP Edition
Manning Early Access Program
Node.js in Action version 6

Copyright 2012 Manning Publications

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

Table of contents

Part 1: Node fundamentals

Chapter 1: Why the web needs Node

Chapter 2: Getting started with Node

Chapter 3: Asynchronous programming

Part 2: Web application development with Node

Chapter 4: Building Node web applications

Chapter 5: Storing Node application data

Chapter 6: Testing Node applications

Chapter 7: Connect

Chapter 8: Connect's built-in middleware

Chapter 9: Express

Chapter 10: Web application templating

Chapter 11: Deploying Node web applications

Part 3: Going further with node

Chapter 12: Beyond web servers

Chapter 13: The Node ecosystem

Appendixes

Appendix A: Installing Node on Windows using Cygwin

Appendix B: Debugging Node applications

Appendix C: Creating documentation

Why the Web needs Node



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

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

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

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

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

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



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

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

1.1 Node is moving the Web forward

Node is the most exciting innovation in web frameworks since Ruby on Rails in 2004. Every once in awhile something new happens in web development which changes the way developers think. Rails came into prominence 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 familiar to most web developers, called Asynchronous JavaScript and XML (AJAX) that allows the browser to send and receive information without requiring page reloads. Combined with Dynamic HTML (DHTML) techniques, which used client-side browser manipulation to create rich interfaces, AJAX made possible the first generation of quasi-real-time web applications. The launching of Google's AJAX-driven "Maps" application in 2005 further drove home the potential of the Web for interactivity.

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

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

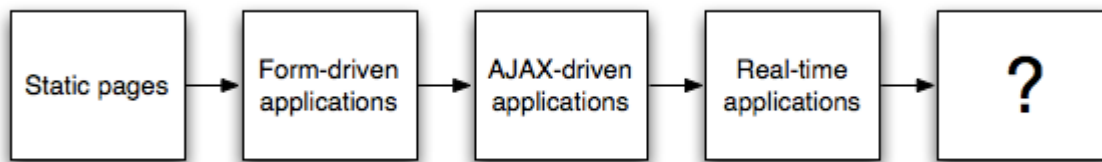


Figure 1.2 A short history of Web technology. As immediacy of interactivity increases, web pages become more like traditional desktop applications that don't require installation. With Node making real-time interactivity accessible, who knows where the Web will go next?

1.1.2 Real-time Web with Node

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

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

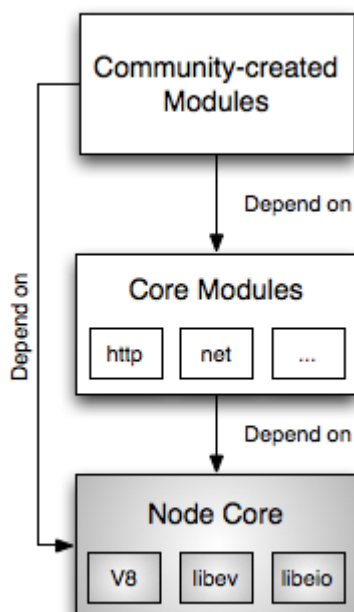


Figure 1.3 Node is conceptually composed of three layers of functionality: the underlying core engine, a number of core modules that add provide utility functions and APIs, and community-created modules for everything else.

V8 was created primarily to provide a fast JavaScript engine for Google's Chrome browser, but V8's developers open sourced it with the intent that it be used by other projects as well. V8's primary innovation is that, unlike traditional JavaScript engines, it compiles, rather than interprets, JavaScript. Compilation in V8 is done on-the-fly, without causing a noticeable 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 minimalist design choices

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

Simplicity is prerequisite for reliability

-- Edsger W. Dijkstra

STARTING FROM SCRATCH

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

SIMPLE, YET POWERFUL, APIS

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

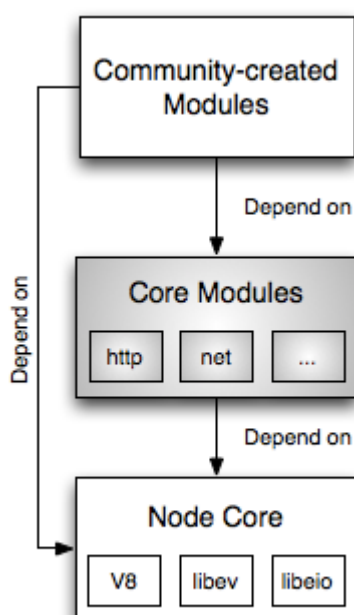


Figure 1.4 Node's core modules provide a wide range of functionality.

SIMPLER HANDLING OF TASK DELEGATION

Node's preference for simplicity, however, extends beyond the API. Most established web application platforms rely on "threads". Think of threads as computational workspaces in which the processor works on one task. In many cases, a thread is contained inside a process and maintains its own working memory. Each thread handles one or more server connections. While this sounds like a natural way to delegate server labor, to people who've been doing this a long time, managing threads within an application can be complex. Also, when a large number of threads are needed to handle many concurrent server connections, threading can tax operating system resources. Each thread requires CPU resources to schedule when it can work and each requires a certain amount of RAM.

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

MINIMALIST ARCHITECTURE

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

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

1.2.2 The asynchronous advantage

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

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

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

As an example of this technique in action, imagine a web application which allows the user to manage a to-do list, storing to-dos in a database. When using the web application, a user could submit a form, specifying a task, to the application and the application could, before even completing the insertion of the task into a database, return a web page to the user. Studies have shown² that response speed is the most important factor to users and a quick web application response can do a lot to increase a website's "stickiness".

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

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

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

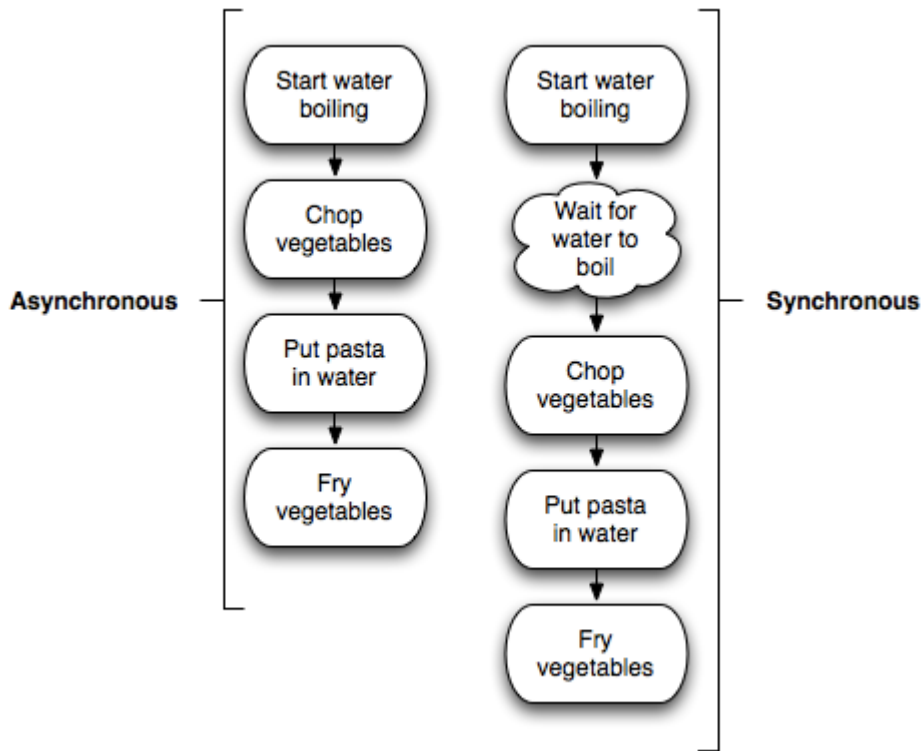


Figure 1.5 Using cooking as an example, asynchronous execution is shown to be more efficient because it requires less time be spent waiting around. Node-based asynchronous development results in efficient applications.

You say you're not a cook? Lets contrast 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 do 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, let's talk about why JavaScript is such a good fit for this style of programming.

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

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

-- Ryan Dahl, creator of the Node project

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

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

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

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

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

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

```
function horse_race(success, failure) {  
  
    var victory = Math.round(Math.random());  
  
    if (victory) {  
        success();  
    } else {  
        failure();  
    }  
}  
  
horse_race(  
    function() { console.log('Goodness! I won!'); },  
    function() { console.log('Drat. I lost.')}  
);
```

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

NOTE **Monkey patching**

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

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

1.2.4 Advancing the idea of the application server

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

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

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

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

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

NOTE**MVC**

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

Footnote 3

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

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

1.3 Node can do more

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

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

1.3.1 Server applications

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

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

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

Browserling, shown in figure 1.6, is an example of a Node-based website that makes great use of Node's non-web capabilities behind the scenes. The site allows in-browser use of other browsers, including the notorious Internet Explorer 6. This is extremely useful to front-end web developers as it frees them from having to install numerous browsers and operating systems solely for testing. Browserling leverages a Node-driven project called StackVM which manages virtual machines (VMs), created using the QEMU ("Quick Emulator") emulator. QEMU emulates

the CPU and peripherals needed to run the browser. Browserling has VMs run test browsers and StackVM then relays video and keyboard/mouse input data between the user's browser and the emulated machines.

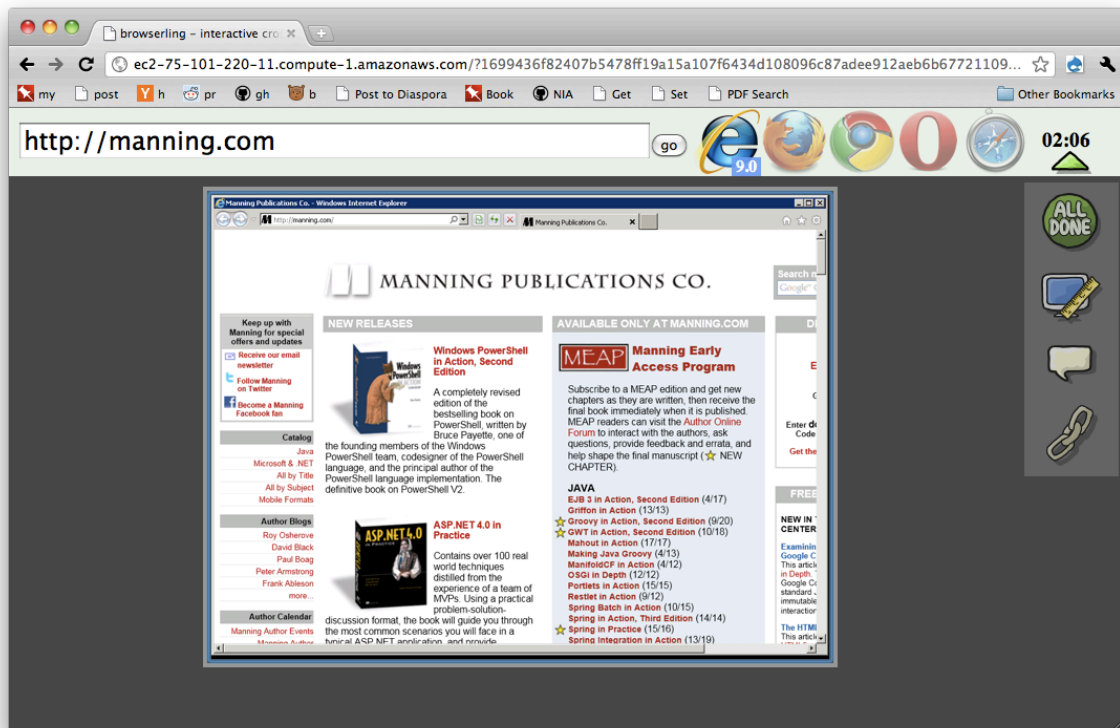


Figure 1.6 Browserling is a web application, predominantly used for cross-browser testing, that uses the Node-driven StackVM server application to manage virtual machines running browsers.

1.3.2 Protocol development

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

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

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

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

1.3.3 *Command-line applications*

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

The Node community has created modules that make CLI application 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.