DZone Refcardz

CONTENTS INCLUDE:

Node.js: Building for Scalability with Server-Side JavaScript

By Todd Eichel

WHAT IS NODE?

In its simplest form, Node is a set of libraries for writing high-performance, scalable network programs in JavaScript. Take a look at this application that will respond with the text "Hello world!" on every HTTP request:

```
// require the HTTP module so we can create a server object
var http = require('http');

// Create an HTTP server, passing a callback function to be
// executed on each request. The callback function will be
// passed two objects representing the incoming HTTP
// request and our response.
var helloServer = http.createServer(function (req, res) {

  // send back the response headers with an HTTP status
  // code of 200 and an HTTP header for the content type
  res.writeHead(200, {'Content-Type': 'text/plain'});

  // send back the string "Hello world!" and close the
  // connection
  res.end('Hello world!');
});

// tell our hello world server to listen for HTTP requests
// on localhost's port 8124
helloServer.listen(8124, "127.0.0.1");

// log a message to the console
console.log('Server running at http://127.0.0.1:8124/');
```

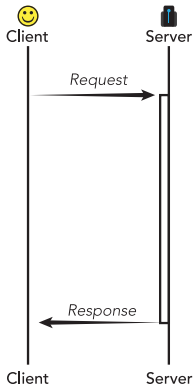
By itself, Node provides only very simple, low-level functionality. However, several external factors have created excitement and interest around it:

1. Developers are increasingly focused on scalability. Node's asynchronous programming model is well suited to building highly scalable web applications.
2. JavaScript is naturally asynchronous, being born and developed inside web browsers.
3. A huge base of developers are already familiar with both JavaScript and asynchronous programming from years developing JavaScript in web browsers.
4. Huge advances in execution speed have made it practical to write server-side software entirely in JavaScript.

Let's take a closer look at the qualities of Node that make these four things possible.

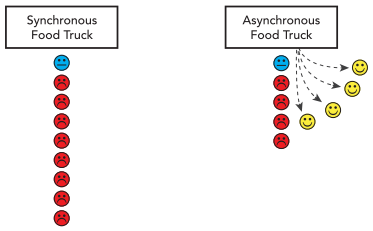
Asynchronous

Node's speed and scalability comes largely from its asynchronous programming model. Traditional web application software is based around the request-response cycle. A request arrives; the application routes it, builds up a response by consulting databases, disks, APIs, etc., and finally sends the completed response back to the client in one piece.



This monolithic approach is very simple but leaves a lot of potential performance on the table. This is easiest to illustrate with a real-world analogy.

Consider a food vending truck on a city street or at a festival. A food truck operating like a traditional synchronous web server would have a worker take an order from the first customer in line, and then the worker would go off to prepare the order while the customer waits at the window. Once the order is complete, the worker would return to the window, give it to the customer, and take the next customer's order.



Contrast this with a food truck operating like an asynchronous web server. The workers in this truck would take an order from the first customer in line, issue that customer an order number, and have the customer stand off to the side to wait while the order is prepared. The worker and window is freed to take the next customer's order. When orders are finished, customers are called back to the window by their number to pick them up.

In the real world, food trucks operate on the asynchronous model. It's the clear choice for efficiency and effective use of resources, at the cost of a little extra complexity with calling people back to the window for pick up. Node works the same way, with similar tradeoffs. Node strives to **never block** subsequent code from executing, just like a real-world food truck operator would never hold a customer at the window waiting for their order, blocking other customers.

Server Side JavaScript

Asynchronous programming is not a new concept in computer science by any means, and many asynchronous web servers are in use in all corners of the web (e.g., Ruby has EventMachine, Python has Twisted). But Node has one thing that sets it apart from the others: JavaScript.

JavaScript is an inherently asynchronous language. It was born

in the web browser, where making blocking calls meant holding up the rendering of web pages or responses to user actions. As a result, JavaScript has no standard library full of blocking file I/O functions or network code. This is where Node comes in—to provide all that functionality we'd expect in a programming language in a completely asynchronous, non-blocking way.

In contrast, normal synchronous languages have standard libraries and open-source packages chock full of blocking code. Writing an asynchronous web server in a synchronous language makes it really easy to trip up and call a blocking function, stopping your web server process cold while the blocking call completes. In Node, it's impossible to accidentally do this because everything has been written from the ground up to be asynchronous.

Common Language

There is a certain elegance to writing both the server- and client-side application code in the same language. Doing this makes it simple to share data back and forth, and it becomes natural to share code as well. This decreases the cost of development and maintenance for shared functionality (e.g., custom string formatting functions).

Speed

Server Side JavaScript owes its growing prominence largely to recent advances in the speed of JavaScript engines. Over the past few years, browser makers like Google and Mozilla have been making huge investments in their JavaScript engines to improve the speed of both their browsers and the web applications running within them. As software is increasingly delivered over the web, JavaScript execution speed has become a larger and larger factor in application performance. The competition between browser makers over JavaScript execution speed has resulted in many orders of magnitude of improvement, to the point that JavaScript now beats many other interpreted languages in common benchmarks[1]. The result of this has been increased interest in running JavaScript on the server, which Node enables in a practical and useful way.

Developer Familiarity

Almost any developer working with web software will have touched JavaScript at some point in his career. JavaScript has been around since the dawn of the web in the mid-nineties, and at least some JavaScript is present in almost every non-trivial web site. This has yielded a huge base of JavaScript-savvy developers already familiar with working in asynchronous programming models. Node is beginning to blur the lines between front-end and back-end developers, and some are starting to cross over from the front-end to the server-side. Developing JavaScript in the browser turns out to be excellent preparation for writing highly scalable network (and web) applications.

See the following two examples. In the first, we have client-side code setting up a callback to be executed on every click of a button. On each click, we print the message "Hello, button clicker!". In the second example, we have Node code setting up an HTTP server to execute a function on every request. Every time an HTTP request comes in, we respond with the message "Hello, web request maker!".

```
document.getElementById('helloButton').onclick = function () {
  alert('Hello, button clicker!');
};

http.createServer(
  function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end("Hello, web request maker!");
  }
).listen(8124);
```

This is a powerful demonstration of how developer familiarity with asynchronous programming in JavaScript on the client side translates to the server side. Any developer who has set up an event handler in the browser should be immediately familiar with Node's asynchronous way of doing things.

WHERE DOES NODE FIT?

Where does Node fit into the web development ecosystem? Where should you use it? What is it good at? What can you do with Node that you can't already do now?

Node's specialty is high-concurrency, real-time applications: anything where you need to have a large number of users connected at the same time. But Node can be used to serve any website or application, and it will do so with its characteristic speed and efficiency.

A couple of years ago, the explosion in popularity of MVC web frameworks like Rails and Django opened up the possibility of building database-backed applications to a whole new class of developers who had never been able to do that before. Those frameworks made that type of application development accessible to a large number of people who before wouldn't have known where to start.

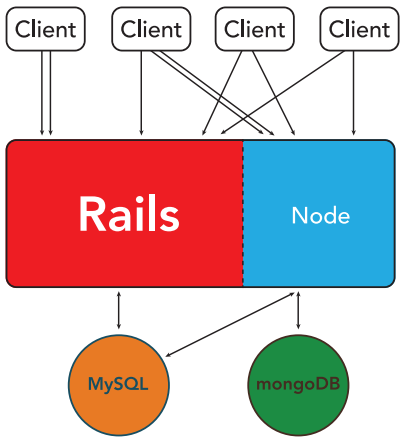
Node is doing the same thing for real-time, high concurrency applications. The kind of highly concurrent programs that can now be written quickly and easily in Node—until recently the sole province of hardcore network programmers—can now be written by any JavaScript-savvy web developer.

Fundamentally, it is important to understand that Node exists on a different level of the stack than web frameworks like Rails; it's actually much closer to the language level than the framework level. Those who try to use it like a framework will walk away disappointed (they should try out Express at http://expressjs.com, a web micro-framework written in Node which we'll cover later). Node is not the new Rails or the new PHP. It is a different tool entirely, and it actually fits in well as a complement to existing tools.

Hybrid Apps

Currently, a common practice is to use Node to supplement existing applications where real-time features are desired. Though it is not impossible to add those types of features using existing tools, Node is so well suited to the task that it's a natural fit.

A typical scenario involves an existing monolithic application with a small additional feature added on being served by a Node application. Typical features to be served this way are chat functionality (like Facebook or Gmail chat) or real-time push notifications (which sites like Quora use extensively). In this scenario, the clients interact with both sides simultaneously, and both sides are probably accessing the same datastores. The main application features are served by the existing application, and the real-time features are served by a small Node application running alongside.



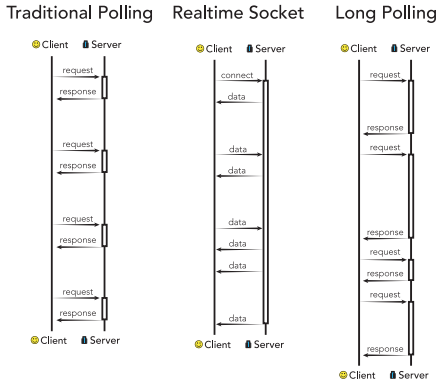
Node Specialties

Node handles some real-world examples in web architecture completely naturally, but these examples are very difficult or impossible to do with existing frameworks. These are where Node can really shine when added to an existing application or used standalone.

Push Notifications

As the web moves towards more highly interactive web sites, a common feature to add to them is push notifications. Push notifications are a hallmark of the emerging real-time web. This is something that is very difficult to implement inside of traditional synchronous application frameworks.

The way push notifications are implemented now is a bit of a stop-gap measure. There are three main ways to get fresh data from the server to the client, illustrated below.



The first is traditional polling. The client makes a request to the server on an interval, asking for any new data. This can only be as real-time as the polling interval, but decreasing the polling interval results in increased server load as every client hits the server on that interval even if there is no new data for them. This model does not scale well and is not commonly used for high-traffic sites.

The second is the holy grail of client-server communications: the real-time socket. A persistent, full-duplex, bi-directional channel is opened between the client and the server, and both can push data through at any time with almost no overhead (compared to HTTP). This is the goal of the WebSocket protocol. The trouble with this method is that most clients and many web servers do not support it currently. While WebSocket support is improving, it represents a big shift in the way the web works and it will be a long time before we can depend on it as developers working in the real world.

So neither of our first two models for implementing push notifications actually work in the real world. Traditional polling is too heavy on the server and doesn't get data out fast enough. Real-time sockets are perfect for this but aren't supported by most browsers.

The third and final model is long polling. This is actually the way most of the real-time web currently works. Clients are instructed to always open a request to the server waiting for new data. If the server doesn't have anything, it holds the request open and doesn't return it until it has a piece of data to push out. Upon receiving data, the client always immediately reopens a new connection to the server waiting for more data. This is an effective way to implement push notifications; however, the cost is that for each client using the site, a connection on the server must be kept open indefinitely. Since most traditional web frameworks can only support one client connection per server thread, this can get expensive quickly in terms of CPU and memory usage. Node, however, can handle many thousands of concurrent client connections with a single server process.

Node's ability to transparently handle huge numbers of concurrent connections makes it perfectly suited for implementing long polling-based push notifications in a web application, where other synchronous frameworks struggle.

Streaming Responses

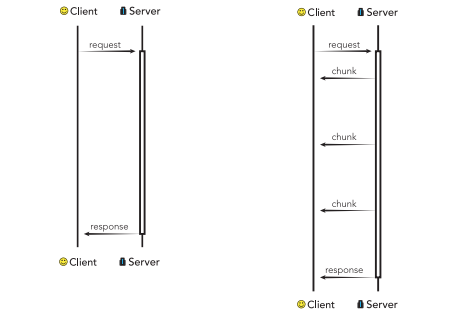
As mentioned before, current web technologies are heavily based around the request-response cycle. The server software receives a request from the client, routes it to the appropriate handler, and begins to render a response. It may consult one or more databases, it might call external APIs, or it may perform some computation. It will then probably render a view template with the resulting data; and only after all that is done, it will ship the finished, rendered response off to the client.

This monolithic approach is easy to develop for but leaves a lot of potential performance benefits sitting on the table. Browsers were built from the beginning to work with partial responses trickling in over slow dial-up connections. Browsers would still start rendering the page as best they could to give the user a good initial experience while still downloading the remainder of the page in the background.

By waiting on the server until the entire response is rendered before returning anything at all, we're leaving this capability untapped. Why shouldn't we first send down the header for our document, which likely contains a bunch of CSS and JavaScript assets that the browser could start loading while we're building the rest of our response?

Streaming responses provides a better way to do this. We can return a little bit of the page at a time, as soon as we have it ready. This is impossible under many existing frameworks, which are entirely built around the request-response cycle. But it's a completely natural model for Node.

Here's an example illustrating streaming responses in Node. As soon as the request comes in, we immediately write out the headers and the first line "Starting...". Then we set up an interval to write the word "data" to the client every half a second. Finally, we set a timeout to end the response with the word "Done!" after five seconds have elapsed. If you connected to this Node application in your browser, you would see the first line immediately and then each subsequent line as it comes in. If you wrote a similar example program in a synchronous language using sleep functions instead of callbacks, the browser would show nothing for the first five seconds and then show the full result all at once.




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

http.createServer(function (request, response) {

  response.writeHead(200, {'Content-Type': 'text/plain'});

  response.write("Starting...\n");

  var dataWriter = setInterval(function () {
    response.write("data\n");
  }, 500);

  setTimeout(function () {
    clearInterval(dataWriter);
    response.end("Done!");
  }, 5000)

}).listen(8124);
```

INSTALLATION

Node

Node falls into the category of software that is under such rapid development that the recommended installation method is to compile it from source code. Pre-packaged binaries do exist, but the recommended method is still to install from source. The following will work on any POSIX-compliant system (Mac OS, Linux, Cygwin on Windows, etc.).

Sidebar: Windows

Node does not run natively on Windows. It does run fine under Cygwin (http://www.cygwin.com/), which is an easily installed, self-contained POSIX environment that runs on top of Windows. Or you can always install Node inside a Linux virtual machine.

If you do choose to run Node under Cygwin, you will need to use the Cygwin package manager to install several additional packages that are not included by default in order for the following commands to work. You'll have an opportunity to do this inside the Cygwin installer. At the time of this writing, the following Cygwin packages need to be selected to install Node and npm (a Node package manager that is explained later):

- python
- openssl
- openssl-devel
- gcc4-g++
- git
- make
- pkg-config
- zlib-devel

Some troubleshooting steps are available on the Node Wiki: https://github.com/joyent/node/wiki/Building-node.js-on-Cygwin-(Windows).

Compiling Node

1. First make sure you have the build prerequisites. You'll need to have a C compiler (like `gcc`) installed if you don't already. Other than that, the only required packages are `python` and `libssl-dev`. Use your OS's package manager to install these before proceeding.
2. Grab the latest version of the source (v0.4.7 as of this writing), extract, and switch into the resulting folder. The latest version can always be obtained from the Node web site (http://nodejs.org/#download).

```
wget http://nodejs.org/dist/node-v0.4.7.tar.gz
tar xzvf node-v0.4.7.tar.gz
cd node-v0.4.7/
```

3. Configure, make, and make install, as with any other package you'd compile from source:

```
./configure
make
make install
```

After you complete these steps, you should have access to the `node` binary from your command line. Try it out with `node -v`, which should print the version number. You can also try running it with no arguments, which will start up an interactive REPL (read-eval-print loop) session where you can type lines of JavaScript and have the result evaluated and printed immediately. In normal use when developing Node programs, you would run this with a filename, e.g. `node myapp.js`.

npm

The other essential piece of software for developing in Node is npm (http://npmjs.org/), the most popular Node package manager. A great number of useful Node libraries and open-source projects are distributed through npm, and you can use npm to distribute your own programs or simply manage their dependencies.

npm offers a one-line installer. Just paste the following into your terminal.

```
curl http://npmjs.org/install.sh | sh
```

You can test it by running `npm -v` to get the version number.

When running npm commands (e.g., to install or uninstall packages), you should always use `sudo` for better security. Because npm packages can execute arbitrary scripts during different parts of their lifecycles (install, uninstall, etc.), npm attempts to downgrade its permissions to the `nobody` user before running any package scripts. Running npm with `sudo` allows it to do this, whereas if you ran them as your own user, the package scripts would execute with whatever permissions you have.

npm's authors recommend setting the following configuration to enforce this security measure:

```
npm config set unsafe-perm false
```

QUICK START

As a way to get started with Node development, we'll demonstrate installing the Express web framework and using it to make a simple web application. Make sure you have both Node and npm installed, as described above.

1. Install Express. It's distributed as an npm package, so it's as easy as:

```
sudo npm install express
```

2. Express also installs a binary, which you can use to generate new skeleton applications. If you've used other frameworks like Rails or Django, this will seem familiar. We'll call our app "demoapp".

```
express demoapp
```

3. You'll see some output as Express generates your application skeleton. Note at the end that it suggests installing Jade, which is a JavaScript templating library along the lines of Ruby's HAML. Express uses Jade for templating by default. We can go ahead and install that with npm:

```
sudo npm install jade
```

4. At this point, you can take a look at the files Express created using your text editor. The main application file is `app.js`, and there are folders for view templates, static files, tests, etc. that you would expect in a web framework. If you take a look at `app.js`, you'll see some app setup and configuration lines, a route handler for the root path (/) that renders the `index.jade` template; and at the end is the call to start up the app server on port 3000.

   Once you're done looking through the code, we can start up the server using Node and take a look at our new application.

```
node app.js
```

5. If you open up `http://localhost:3000/` in your web browser, you should see a simple "Welcome to Express" message. Let's try changing part of this and reloading the page. In `app.js`, find the handler for the root path and change the `title` attribute to "Demo App". If we reloaded the page in the browser, we'd expect to see the change reflected as "Welcome to Demo App". But unless you've also restarted the `node` process, we'll still see the original message. This is a common trap that new Node developers coming from the browser or Rails/Django/etc. worlds experience. While those other more involved frameworks will reload your source files on every request, Node is very simple. So restarting the process is up to you. Libraries do exist to make this easier. See the next section for an example.

NODE ECOSYSTEM

The Node ecosystem is in its infancy and still rapidly evolving, but some important projects have emerged for solving common problems.

Express

http://expressjs.com

Express, as demonstrated in the quick start guide, is a web micro-framework along the lines of Ruby's Sinatra or Python's Bottle.

Socket.IO
http://socket.io
Socket.IO is a library for making real-time web apps easy to write by abstracting away browser differences and incompatibilities. It will automatically select the best available transport mechanism (including WebSocket) supported by both the browser and the server. Use Socket.IO whenever you want to add real-time features to you application or if you're writing something inherently real-time like a game or chat app.

streamline.js, step, flow-js
https://github.com/Sage/streamlinejs
https://github.com/creationix/step
https://github.com/willconant/flow-js
As you start to write more complex asynchronous code, you'll begin to see that the complexity of nested callbacks can easily get out of control. This is especially true in the case of database interactions where you may need to perform operations in sequence or only perform a cleanup operation after several others have completed. Several libraries can help you untangle your callback functions in these situations. Because each one is a little different, the best approach would be to take a look at each one and see which best suits your present situation.

Mongoose
http://mongoosejs.com
Mongoose is an ORM for interacting with a MongoDB database. MongoDB is a popular choice of datastore for Node apps, and Mongoose makes it easy to use.

NowJS
http://nowjs.com
NowJS is an excellent demonstration of the things you can do when the client-side and server-side language are the same. It lets you share functions and variables back and forth between the client and the server as well as call client functions from the server and server functions from the client.

Supervisor
https://github.com/isaacs/node-supervisor
This package solves the problem mentioned at the end of the Quick Start section. When you start a Node program using supervisor (`supervisor myapp.js`), it will monitor your application directory and reload Node on the fly whenever it detects changes. This is an indispensible tool for Node development.

Additional Resources
If you need help with Node development, here are some places you can go.

Official Sites
Homepage: http://nodejs.org
Manual and documentation: http://nodejs.org/docs/v0.4.7/api/
Wiki: https://www.github.com/joyent/node/wiki
Blog: http://blog.nodejs.org/
Source code: https://github.com/joyent/node

Articles and Tutorials
Screencast tutorials for beginners: http://nodetuts.com
Advanced articles by Node community leaders:
http://howtonode.org

Community
Node official mailing list:
http://groups.google.com/group/nodejs
Node Official IRC channel: #node.js on irc.freenode.net
Node User Q&A via StackOverflow:
http://stackoverflow.com/questions/tagged/node.js

NODE API GUIDE

Below is a list of the most commonly used Node modules and how you might typically use them. View the full list in the Node documentation http://nodejs.org/docs/v0.4.7/api/.

Timers
Functions for setting and clearing timeouts and intervals just like you would in a browser.

Process
Use for accessing `stdin`, `stdout`, command line arguments, the process ID, environment variables, and other elements of the system related to the currently-executing Node process.

Utilities
Logging, debugging, and object inspection functions.

Events
Contains the EventEmitter class used by many other Node objects. Defines the API for attaching and removing event listeners and interacting with them.

Buffers
Functions for manipulating, creating, and consuming octet streams, which you may encounter when doing your own network or file system I/O. You likely won't interact with these much if you're only doing web application programming.

Streams
An abstract interface for streaming data which is implemented by other Node objects, like HTTP server requests, and even `stdio`. Most of the time you'll want to consult the documentation for the actual object you're working with rather than looking at the interface definition.

Crypto
Functions for dealing with secure credentials that you might use in an HTTPS connection.

TLS/SSL
Functions for making or serving requests over SSL. See also the HTTPS module.

File System
File system interaction functions. Read/write files and directories, move, copy, rename files. Some functions have synchronous versions alongside the normal asynchronous ones, as noted in their names. These are useful in the setup phases of an application's execution, where speed is unimportant and the simplicity of a synchronous function is desired.

Path
Complements the File System module; provides functions to manipulate paths and filenames, resolve relative paths, etc.

Net
The meat of Node's functionality. Create network server objects to listen for connections and act on them. Read from and write to sockets. Most of the time if you're working on web applications, you won't interact with this directly. Instead you'll use the HTTP module to create HTTP-specific servers. If you want to create TCP servers and sockets and interact with them directly, look here.

UDP/Datagram
Functions for handling UDP servers and messages. If you don't know what UDP is, then you probably don't need to worry about this module.

DNS
Contains functions for doing regular or reverse DNS lookups (e.g. of a domain name to an IP address) and fetching DNS records (e.g. A, CNAME, MX) for a domain.

HTTP
This is the most important and most used module for a web developer. Create HTTP servers and have them listen on a given port. This also contains the request and response objects that hold information about incoming requests and outgoing responses. You can also use this to **make** HTTP requests from your application and do things with their responses.

HTTPS
Contains functions for creating HTTPS servers or requests. This is regular HTTP secured with SSL. See also the TLS/SSL module.

URL
Interact with URLs: parsing, formatting, resolving an absolute URL from a relative URL with a base URL

Query String
Handle parsing or composing query string parameters (including escaping/unescaping strings).

REPL
Short for read-eval-print-loop. You can add a REPL to your own programs just like Node's standalone REPL (which you get if you run node with no arguments).

VM
Allows you to compile arbitrary JavaScript code and optionally execute it new sandboxed context.

Child Processes
Functions for spawning new processes and handling their input and output.

Assertion Testing
Basic facilities for writing and running unit tests.

TTY
Functions for interacting with TTYs. This will probably only be useful to you if you're writing Node programs to be run on the console (e.g. devops or system administration scripts), rather than accessed over the web via HTTP requests (web apps).

OS
Get information from the operating system. Your hostname, OS name and type, uptime, load averages, memory, CPUs.

Debugger
You can access the V8 engine's debugger with Node's built-in client and use it to debug your own scripts. Just launch node with the debug argument (node debug server.js). See the documentation for more usage details.

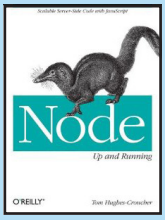
[1] http://shootout.alioth.debian.org/u32/benchmark.php?test=all&lang=v8&lang2=yarv

ABOUT THE AUTHOR

Todd Eichel is a co-founder of Duostack, a cloud platform for Node and Ruby applications. Prior to Duostack, Todd led the development of a Pittsburgh-based local e-commerce company, where he intensively developed his JavaScript and Ruby expertise. Todd has a Master of Information Systems Management degree from Carnegie Mellon University, where he also earned his B.S. in Information Systems with an additional major in Statistics.

When Todd isn't working on apps in the cloud, he's flying through the clouds as a recreational pilot. He currently resides in San Francisco. Find him online at http://toddeichel.com/, on Twitter at http://twitter.com/toddeichel, or on GitHub at https://github.com/tfe.

RECOMMENDED BOOK



Written by a core contributor to the framework, *Node: Up and Running* shows you how Node scales up to support large numbers of simultaneous connections across multiple servers, and scales down to let you create quick one-off applications with minimal infrastructure. Built on the V8 JavaScript engine that runs Google Chrome, Node is already winning the hearts and minds of many companies, including Google and Yahoo! This book shows you why.

Pre-order it now!

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-41-7
ISBN-10: 1-936502-41-0
50795
9 781936 502417
$7.95

Version 1.0