

Just what is Node.js?

A ready-to-code server

Skill Level: Intermediate

[Michael Abernethy](#)
Programmer
Freelancer

26 Apr 2011

Updated 24 May 2011

Node is a server-side Javascript interpreter that changes the notion of how a server should work. Its goal is to enable a programmer to build highly-scalable applications and write code that handles tens of thousands of simultaneous connections on one, and only one, physical machine.

Introduction

If you've heard about Node, or read any articles claiming how awesome it is, you may be wondering, "What is Node.js anyway?". You may even *still* have questions about what Node is after reading their own home page. Node is definitely not for every programmer, but it could be the right answer for some.

This article will seek to answer what Node.js is by giving a brief background of the problems it is solving, how it works, how to run a simple application, and finally, where Node is a good solution. It will not cover how to write a complicated Node application nor be a thorough tutorial on Node. Reading this article should help you decide whether you should further pursue learning Node to use in your own business.

What problem does Node solve?

Node's stated number one goal is "to provide an easy way to build scalable network programs". What's the issue with current server programs? Let's do the math. In languages like Java™ and PHP, each connection spawns a new thread that potentially has an accompanying 2 MB of memory with it. On a system that has 8 GB of RAM, that puts the theoretical maximum number of concurrent connections at about 4,000 users. As your client-base grew, if you wanted your web application to support more users, you had to add more and more servers. Of course, this adds to a business's server costs, traffic costs, labor costs, and more. Adding to those costs are the potential technical issues — a user can be using different servers for each request, so any shared resources have to be shared across all the servers. For all these reasons, the bottleneck in the entire web application architecture (including traffic throughput, processor speed, and memory speed) was the maximum number of concurrent connections a server could handle.

Node solves this issue by changing how a connection is made to the server. Instead of spawning a new OS thread for each connection (and allocating the accompanying memory with it), each connection fires an event run within the Node engine's process. Node also claims that it will never deadlock, since there are no locks allowed, and it doesn't directly block for I/O calls. Node claims that a server running it can support tens of thousands of concurrent connections.

So, now that you have a program that can handle tens of thousands of concurrent connections, what can you actually build with Node? It would be awesome if you had a web application that required this many connections. That's one of those "if you have this problem, it's not a problem" kind of problems. Before we get to that, let's look at how Node works and how it's designed to run.

What Node definitely isn't

Yes, Node is a server program. However, the base Node product is definitely *not* like Apache or Tomcat. Those servers are basically ready-to-install server products and are ready to deploy apps instantly. You could be up and running with a server in a minute with these products. Node is definitely not this. Similar to how Apache can add a PHP module to allow developers to create dynamic web pages, and an SSL module for secure connections, Node has the concept of modules that can be added to the Node core as well. There are literally hundreds of modules to choose from with Node, and the community is quite active in producing, publishing, and updating dozens of modules a day. We'll talk about the entire module part of Node later in the article.

How Node works

Node itself runs V8 JavaScript. Wait...what? JavaScript on the server? Yes, you read that correctly. Server-side JavaScript may be a new concept to everyone who's

worked exclusively with JavaScript on the client, but the idea itself isn't that far fetched — why not use the same programming language on the client that you use on the server?

What's the V8? The V8 JavaScript engine is the underlying JavaScript engine that Google uses with their Chrome browser. Few people think about what actually happens with JavaScript on the client. Well, a JavaScript engine actually interprets the code and executes it. With V8, Google created an ultra-fast interpreter written in C++, with another unique aspect: you can download the engine and embed it in *any* application you wish. It's not restricted to running in a browser. So, Node actually uses the V8 JavaScript engine written by Google and repurposes it for use on the server. Perfect! Why create a new language when there's a good solution already available.

Event-driven programming

Many programmers have been taught to believe that object-oriented programming is the perfect programming design and to use nothing else. Node utilizes what's called an event-driven programming model.

Listing 1. Event-driven programming on client-side with jQuery

```
// jQuery code on the client-side showing how Event-Driven programming works

// When a button is pressed, an Event occurs - deal with it
// directly right here in an anonymous function, where all the
// necessary variables are present and can be referenced directly
$("#myButton").click(function(){
    if ($("#myTextField").val() != $(this).val())
        alert("Field must match button text");
});
```

The server-side is actually no different than the client-side. True, there are no buttons getting pressed, and no text fields getting typed in, but on a higher level, events *are* taking place. A connection is made — event! Data is received through the connection — event! Data stops coming through the connection — event!

Why is this type of setup ideal for Node? JavaScript is a great language for event-driven programming, because it allows anonymous functions and closures, and more importantly, the syntax is familiar to nearly everyone who has ever coded. The callback functions that are called when an event occurs can be written in the same spot where you capture the event. Easy to code, easy to maintain. No complicated Object Oriented frameworks, no interfaces, no potential for over-architecting anything. Just listen for an event, write a callback function, and everything is taken care of!

Example Node application

Let's finally see some code! Let's put together all the things we've discussed and create our first Node application. Since we've seen that Node is ideal for handling high traffic applications, let's create a very simple web application that's built for maximum speed. Here are the specs for our sample application passed down from the "boss": Create a random number generator RESTful API. The application should take one input, a parameter called "number." The application will then return a random number that's between 0 and this parameter, and return that generated number to the caller. And, since the "boss" expects this to be a massively popular application, it should handle 50,000 concurrent users. Let's look at the code:

Listing 2. Node random number generator

```
// these modules need to be imported in order to use them.
// Node has several modules. They are like any #include
// or import statement in other languages
var http = require("http");
var url = require("url");

// The most important line in any Node file. This function
// does the actual process of creating the server. Technically,
// Node tells the underlying operating system that whenever a
// connection is made, this particular callback function should be
// executed. Since we're creating a web service with REST API,
// we want an HTTP server, which requires the http variable
// we created in the lines above.
// Finally, you can see that the callback method receives a 'request'
// and 'response' object automatically. This should be familiar
// to any PHP or Java programmer.
http.createServer(function(request, response) {

    // The response needs to handle all the headers, and the return codes
    // These types of things are handled automatically in server programs
    // like Apache and Tomcat, but Node requires everything to be done yourself
    response.writeHead(200, {"Content-Type": "text/plain"});

    // Here is some unique-looking code. This is how Node retrieves
    // parameters passed in from client requests. The url module
    // handles all these functions. The parse function
    // deconstructs the URL, and places the query key-values in the
    // query object. We can find the value for the "number" key
    // by referencing it directly - the beauty of JavaScript.
    var params = url.parse(request.url, true).query;
    var input = params.number;

    // These are the generic JavaScript methods that will create
    // our random number that gets passed back to the caller
    var numInput = new Number(input);
    var numOutput = new Number(Math.random() * numInput).toFixed(0);

    // Write the random number to response
    response.write(numOutput);

    // Node requires us to explicitly end this connection. This is because
    // Node allows you to keep a connection open and pass data back and forth,
    // though that advanced topic isn't discussed in this article.
    response.end();

    // When we create the server, we have to explicitly connect the HTTP server to
    // a port. Standard HTTP port is 80, so we'll connect it to that one.
}).listen(80);

// Output a String to the console once the server starts up, letting us know everything
```

```
// starts up correctly
console.log("Random Number Generator Running...");
```

Starting this application

Put the above code into a file called "random.js". Now, to start this application and run it (thereby creating the HTTP server and listen for connections on port 80), simply run the following command in your command prompt: `% node random.js`. Here's what it will look like when you know the server is up and running.

```
root@ubuntu:/home/moila/ws/mike# node random.js
Random Number Generator Running...
```

Accessing this application

The application is up and running. Node is listening for any connections right now, so let's test the application. Since we've created a simple RESTful API, we can access the application using our web browser. Type in the following address (make sure you completed the previous step), `http://localhost/?number=27`.

Your browser window will change to a random number between 0 and 27. Press reload on your browser and you'll get another random number. That's it, there's your first Node application!

Node, what is it good for?

So, after reading all about Node, you may be able to answer the question "Just what is Node?" but it may leave you with the question "What should I use Node for?" That's an important question to ask, because there are certain things that Node is really good for.

What it's good for

As you've seen so far, Node is extremely well-designed for situations where you are expecting a high amount of traffic and the server-side logic and processing required isn't necessarily large before responding to the client. Here are some good examples of where Node would excel:

- **A RESTful API**

A web service that provides a RESTful API takes in a few parameters, interprets them, pieces together a response, and flushes a response (usually a relatively small amount of text) back to the user. This is an ideal situation for Node, because it can be built to handle tens of thousands of connections. It also doesn't require a large amount of logic,

and basically just looks up values from a database and pieces together a response. Since the response is a small amount of text, and the incoming request is a small amount of text, the traffic volume isn't high, and one machine can likely handle the API demands of even the busiest company's API.

- **Twitter queue**

Think about a company like Twitter that has to receive tweets and write them to a database. There are literally thousands of tweets arriving every second, and the database can't possibly keep up with the number of writes required during peak usage times. Node becomes an important cog in the solution to this problem. As we've seen, Node can handle tens of thousands of incoming tweets. It can then quickly/easily write them to an in-memory Queuing mechanism (memcached for example), from which another separate process can write them to the database. Node's role in this is to quickly gather the tweet and pass this information off to another process responsible for writing it. Imagine another design — a normal PHP server that tries to handle writes to the DB itself — every tweet would cause a small delay as its written to the DB since the DB call would be blocking. A machine with this design may only be able to handle 2000 incoming tweets a second, due to the database latency. A million tweets a second, and you're talking 500 servers. Node, instead, handles every connection and doesn't block, enabling it to capture as many tweets as can be thrown at it. A node machine able to handle 50,000 tweets a second, and you're talking only 20 servers.

- **Video game statistics**

If you've ever played a game like Call of Duty online, some things stick out at you immediately when you look at the game statistics, mainly that they must be tracking a TON of info about the game in order to produce that level of statistics. Then, factor in the millions of people playing the game at any point, and you realize there is a TON of information getting generated very quickly. Node is a good solution for this scenario, because it can capture the data getting generated from the games, do a minimal amount of consolidation on them, and then queue them up for writing to a database. It would seem silly to devote an entire server to tracking how many bullets people are firing in games, which might be the useful limit if you used a server like Apache, but it would seem less silly if instead you could devote a single server to tracking almost all the statistics from a game, like you might be able to do with a server running Node.

Node modules

While originally not a planned discussion in the article, due to popular demand, the article has been expanded to include a brief introduction to Node Modules and the Node Package Manager. Like people have grown accustomed to when working with Apache, you can expand the functionality of Node by installing modules. However, the modules you can use with Node *greatly* enhance the product, so much so, that it is unlikely that anyone would use Node without installing a few modules as well. That's how great the modules have become, to the point of becoming an essential part of the entire product.

In the Resources, I provide a link to the module page, where all the possible modules are listed and available. As a quick sampling of the possibilities, they include a module to write dynamically created pages (like PHP), a module to ease working with MySQL, a module to help with WebSockets, and a module to assist in text and parameter parsing, among the dozens of available modules. I won't go into the details of the modules, since again, this is just an overview article helping you to understand if Node is something you should pursue further, but chances are, if you choose to pursue it further, you will definitely also be working with the modules available.

Additionally, Node feature the Node Package Module, which is a built-in way to install and manage the Node modules you are using. It automatically handles dependencies, so you can be assured that any module you want to install will install correctly with all its required pieces. It also serves as a way to publish your own modules to the Node community, should you choose to get involved and write your own module. Think of NPM as a way to easily expand the functionality of Node without worrying about breaking your Node installation. Again, if you choose to pursue Node further, the NPM is going to be a vital part of your Node solution.

Conclusion

Editor's note

The initial published version of this article generated a lot of conversation from the community about various points it presented. The author has since revised this article with these ideas in mind. This kind of peer review and discussion is a vital part of the open-source world. Thanks to those who provided constructive input.

Like all open-source projects, Node.js will continue to evolve and developers will discover new resources and techniques to overcome any number of limitations. As always, we encourage our readers to try a technology out for themselves.

The question many of you had at the beginning of this article "Just what is Node.js?" should be answered after reading this article. You should be able to explain in a few clear and concise sentences what Node.js is. If you are able to do that, you have a

leg up on almost every other programmer out there. Many people I've talked to about Node have been confused by what it does exactly. They are, understandably, in the Apache mindset — a server is an application that you drop your HTML files into and everything works. Since most programmers are used to Apache and what it does, the easiest way to describe Node is to compare it to Apache. Node is a program that can do anything Apache can do (with some modules), but is also able to do much more, by being an extend-able JavaScript platform that you can build from.

You've seen in this article how Node accomplishes its goals of providing highly-scalable servers. It uses an extremely fast JavaScript engine from Google, the V8 engine. It uses an Event-Driven design to keep code minimal and easy-to-read. All of these factors lead to Node's desired goal — it's relatively easy to write a massively-scalable solution.

Just as important as understanding what Node *is*, it's also important to understand what it *is not*. Node is not simply a replacement for Apache that will instantly make your PHP web application more scalable. That couldn't be further from the truth. It's still early in Node's life, but it's growing extremely quickly, the community is very actively involved, there are a lot of good modules being created, and this growing product could be in your business within a year.

Resources

Learn

- The [Node.js home page](#) is the launching point for learning about the application.
- [Download Node.js here](#). You'll need [Python](#) too.
- Browse [the Node.js API page](#). Note that the syntax may change from release to release, so double check the version you've downloaded and the API you're browsing.
- See [the Node module page](#) that lists all the available modules for use in Node.
- Search the [NPM](#) to make expanding your Node install's functionality easy.
- Stay current with [developerWorks technical events and webcasts](#).
- Stay current with developerWorks' [Technical events and webcasts](#).
- Follow [developerWorks on Twitter](#).
- Check out upcoming conferences, trade shows, webcasts, and other [Events](#) around the world that are of interest to open source developers.
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM products.
- Learn about IBM and open source technologies and product functions with the no-cost [developerWorks on-demand demos](#).

Get products and technologies

- See the ever-evolving list of [open source software packages](#) on Wikipedia.
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- [Participate in the discussion forum](#).
- Help build the [Real world open source](#) group in developerWorks Community.

About the author

Michael Abernethy



Michael Abernethy has worked with a wide variety of technologies and a wide variety of clients. He focuses on building better and more complex web applications, testing the limits of the browsers they run on, and trying to figure out how to make them easier to create and easier to maintain. When he's not working at his computer, he can be found hanging out with his kids with a good book.