JavaScript Breaks Out of the Browser

What Is Node?

O'REILLY®

Brett McLaughlin

What is Node.js?

Brett McLaughlin

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Chapter 1. What is Node.js?

Node isn’t always the solution, but it does solve some important problems.

Learning Node might take a little effort, but it’s going to pay off. Why? Because you’re afforded solutions to your web application problems that require only JavaScript to solve.

By Brett McLaughlin

SECTIONS

Node.js. It’s the latest in a long line of “Are you cool enough to use me?” programming languages, APIs, and toolkits. In that sense, it lands squarely in the tradition of Rails, and Ajax, and Hadoop, and even to some degree iPhone programming and HTML5. Go to a big technical conference, and you’ll almost certainly find a few talks on Node.js, although most will fly far over the head of the common mortal programmer.

Dig a little deeper, and you’ll hear that Node.js (or, as it’s more briefly called by many, simply “Node”) is a server-side solution for JavaScript, and in particular, for receiving and responding to HTTP requests. If that doesn’t completely boggle your mind, by the time the conversation heats up with discussion of ports, sockets, and threads, you’ll tend to glaze over. Is this really JavaScript? In fact, why in the world would anyone want to run JavaScript outside of a browser, let alone the server?

The good news is that you’re hearing (and thinking) about the right things. Node really is concerned with network programming and server-side request/response processing. The bad news is that like Rails, Ajax, and Hadoop before it, there’s precious little clear information available. There will be, in time — as there now is for these other “cool” frameworks that have matured — but why wait for a book or tutorial when you might be able to use Node today, and dramatically improve the maintainability of your code and even the ease with which you bring on programmers?

A warning to the Node experts out there

Node is like most technologies that are new to the masses, but old hat to the experienced few: it's opaque and weird to most but completely usable for a small group. The result is that if you've never worked with Node, you're going to need to start with some pretty basic server-side scripts. Take your time making sure you know what's going on, because while this is JavaScript, it's not operating like the client-side JavaScript you're used to. In fact, you're going to have to twist your JavaScript brain around event loops and waiting and even a bit of network theory.

Unfortunately, this means that if you've been working and playing with Node for a year or two, much of this article is going to seem pedestrian and overly simplistic. You'll look for things like using Node on the client, or heavy theory discussions on evented I/O and reactor patterns, and `npm`. The reality is that while that's all interesting — and advances Node to some pretty epic status — it's incomprehensible to someone just getting started out. Given that, maybe you should pass this piece on to your co-workers who *don't* know Node, and then when they're buying into Node's usefulness, start to bring them along on the more advanced Node use cases.

Node: A few basic examples

First things first: you need to realize that Node is intended to be used for running standalone JavaScript programs. This isn’t a file referenced by a piece of HTML and running in a browser. It’s a file sitting on a file system, executed by the `Node` program, running as what amounts to a daemon, listening on a particular port.

Skipping hello world

The classic example here is “Hello World,” detailed on the Node website. Almost everyone starts with Hello World, though, so check that out on your own, and skip straight to something a lot more interesting: a server that can send static files, not just a single line of text:

```
var sys = require("sys"),
    http = require("http"),
    url = require("url"),
    path = require("path"),
    fs = require("fs");

http.createServer(function(request, response) {
    var uri = url.parse(request.url).pathname;
    var filename = path.join(process.cwd(), uri);
    path.exists(filename, function(exists) {
        if(!exists) {
            response.writeHead(404, {"Content-Type":
"text/plain"});
            response.end("404 Not Found\n");
            return;
        }

        fs.readFile(filename, "binary", function(err, file) {
            if(err) {
                response.writeHead(500, {"Content-Type":
"text/plain"});
                response.end(err + "\n");
                return;
            }

            response.writeHead(200);
            response.end(file, "binary");
        });
    });
}).listen(8080);

console.log("Server running at http://localhost:8080/");
```

Thanks much to Mike Amundsen for the pointer to similar code. This particular example was posted by Devon Govett on the Nettuts+ training blog, although it’s been updated for the current version of Node in a number of places. Devon’s entire tutorial post is actually a great companion piece on getting up to speed on Node once you have a handle on the basics.

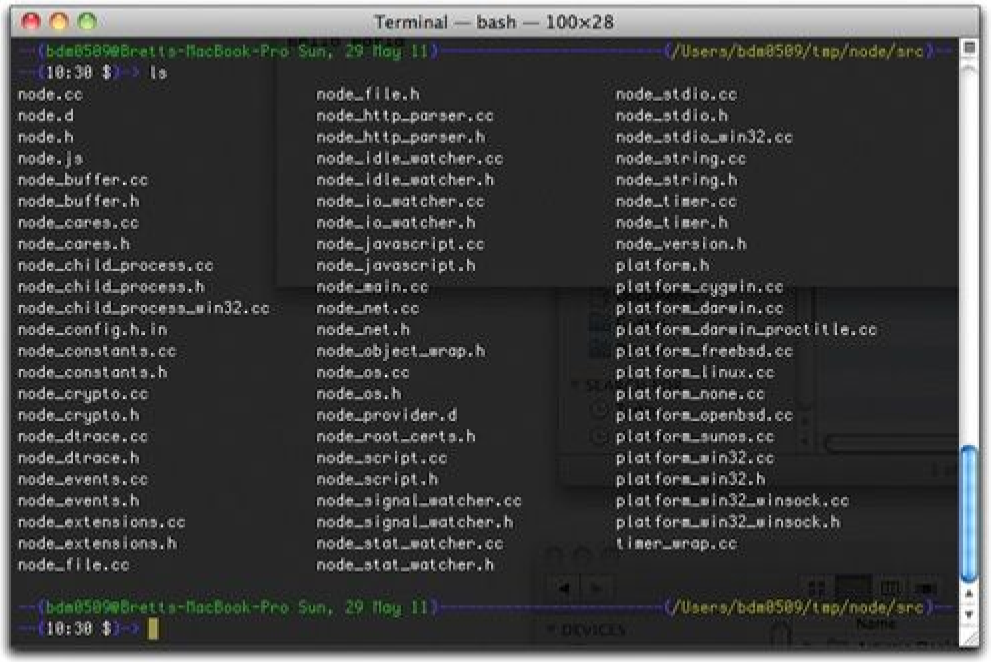
If you’re new to Node, type this code into a text file and save the file as *NodeFileServer.js*. Then head out to the Node website and download Node or check it out from the git repository. You’ll need to build the code from source; if you’re new to Unix, `make`, and `configure`, then check out the online build instructions for help.

Node runs JavaScript, but isn't JavaScript

Don't worry that you've put aside *NodeFileServer.js* for a moment; you'll come back to it and more JavaScript shortly. For now, soak in the realization that you've just run through the classic Unix configuration and build process:

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

That should come with another realization: Node itself isn't JavaScript. Node is a program for *running* JavaScript, but isn't JavaScript itself. In fact, Node is a C program. Do a directory listing on the *Node/src* directory and you'll see something like this:



```
—(bdm0509@Bretts-MacBook-Pro Sun, 29 May 11)——————————(/Users/bdm0509/tmp/node/src)—
—(10:30 $)—> ls
node.cc                          node_file.h                      node_stdio.cc
node.d                           node_http_parser.cc              node_stdio.h
node.h                           node_http_parser.h               node_stdio_win32.cc
node.js                          node_idle_watcher.cc             node_string.cc
node_buffer.cc                   node_idle_watcher.h              node_string.h
node_buffer.h                    node_io_watcher.cc               node_timer.cc
node_cares.cc                    node_io_watcher.h                node_timer.h
node_cares.h                     node_javascript.cc               node_version.h
node_child_process.cc            node_javascript.h                platform.h
node_child_process.h             node_main.cc                     platform_cygwin.cc
node_child_process_win32.cc      node_net.cc                      platform_darwin.cc
node_config.h.in                 node_net.h                       platform_darwin_proctitle.cc
node_constants.cc                node_object_wrap.h               platform_freebsd.cc
node_constants.h                 node_os.cc                       platform_linux.cc
node_crypto.cc                   node_os.h                        platform_none.cc
node_crypto.h                    node_provider.d                  platform_openbsd.cc
node_dtrace.cc                   node_root_certs.h                platform_sunos.cc
node_dtrace.h                    node_script.cc                   platform_win32.cc
node_events.cc                   node_script.h                    platform_win32.h
node_events.h                    node_signal_watcher.cc           platform_win32_winsock.cc
node_extensions.cc               node_signal_watcher.h            platform_win32_winsock.h
node_extensions.h                node_stat_watcher.cc             timer_wrap.cc
node_file.cc                     node_stat_watcher.h

—(bdm0509@Bretts-MacBook-Pro Sun, 29 May 11)——————————(/Users/bdm0509/tmp/node/src)—
—(10:30 $)—>
```

For all of you thinking that JavaScript is a poor language in which to be writing server-side tools, you're half right. Yes, JavaScript is not equipped to deal with operating system-level sockets and network connectivity. But Node isn't written in JavaScript; it's written in C, a language perfectly capable of doing the grunt work and heavy lifting required for networking. JavaScript is perfectly capable of sending instructions to a C program that can be carried out in the dungeons of your OS. In fact, JavaScript is far more accessible than C to most programmers — something worth noting now, and that will come up again and again in the reasons for looking seriously at Node.

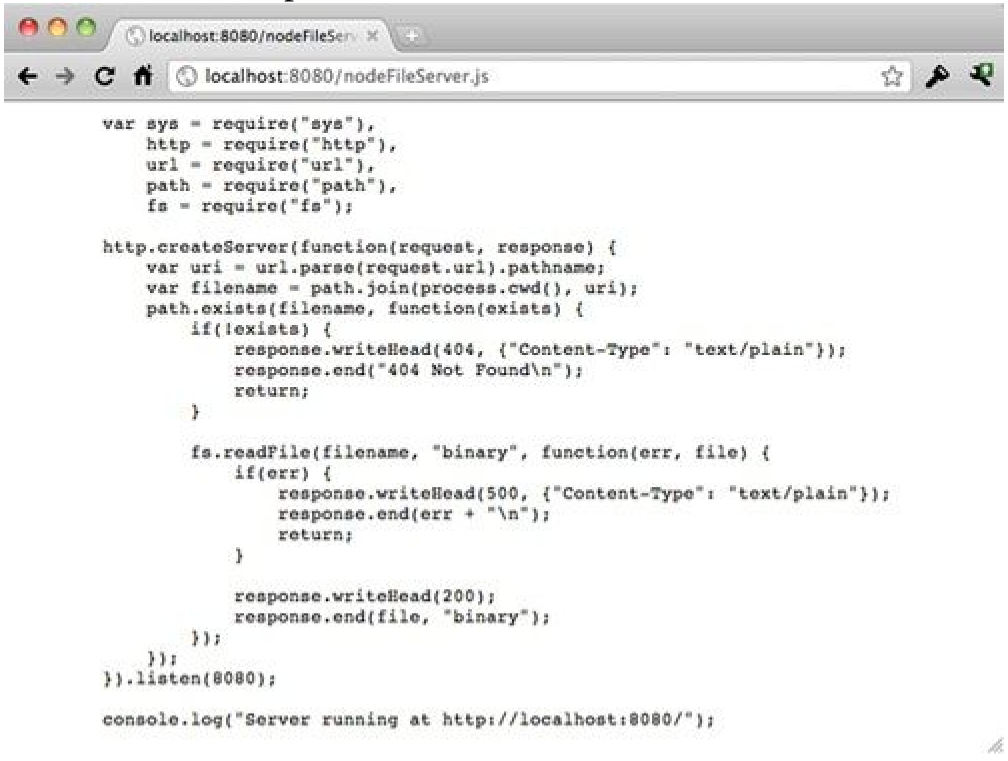
The primary usage of Node further reflects that while Node works with JavaScript, it isn’t itself JavaScript. You run it from the command line:

```
  —  (bdm0509@Bretts-MacBook-Pro Sun, 29 May 11)  —    —    —    —
—    —    —    —    —    —  (/Users/bdm0509/tmp/Node/src)  —
 —  (09:09 $)-> export PATH=$HOME/local/Node/bin:$PATH

 —  (bdm0509@Bretts-MacBook-Pro Sun, 29 May 11)  —    —    —    —
—    —    —    —    —    —  (/Users/bdm0509/tmp/Node/src)  —
 —  (09:09 $)-> cd ~/examples

 —  (bdm0509@Bretts-MacBook-Pro Sun, 29 May 11)  —    —    —    —
—    —    —    —    —    —    —    —  (/Users/bdm0509/examples)
—
 —  (09:09 $)-> Node NodeFileServer.js
Server running at http://127.0.0.1:1337/
```

And there you have it. While there’s a lot more to be said about that status line — and what’s really going on at port 1337 — the big news here is that Node is a program that you feed JavaScript. What Node then does with that JavaScript isn’t worth much ink; to some degree, just accept that what it does, it *does*. This frees you up to write JavaScript, not worry about learning C. Heck, a big appeal to Node is that you can actually write a server without worrying about C. That’s the point.

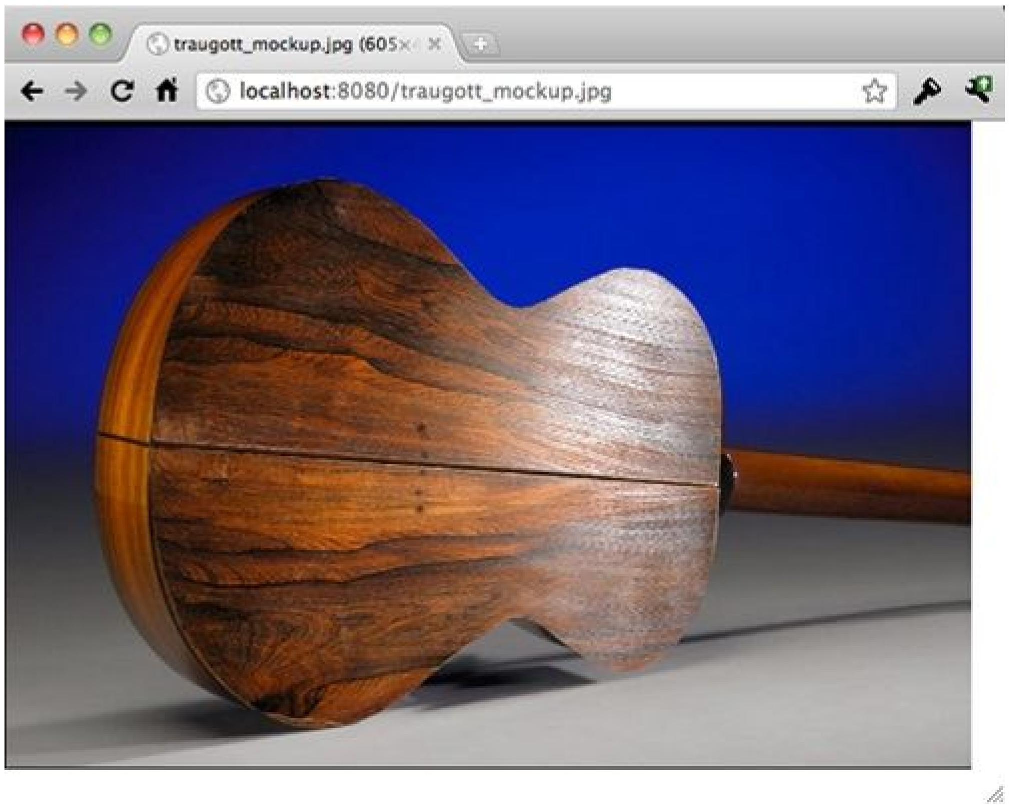
Interacting with a “Node server”

Make sure you still have your *NodeFileServer.js* code running via Node. Then you can hit your local machine — on port 1337 — and see this unremarkable output.



```
var sys = require("sys"),
    http = require("http"),
    url = require("url"),
    path = require("path"),
    fs = require("fs");

http.createServer(function(request, response) {
    var uri = url.parse(request.url).pathname;
    var filename = path.join(process.cwd(), uri);
    path.exists(filename, function(exists) {
        if(!exists) {
            response.writeHead(404, {"Content-Type": "text/plain"});
            response.end("404 Not Found\n");
            return;
        }

        fs.readFile(filename, "binary", function(err, file) {
            if(err) {
                response.writeHead(500, {"Content-Type": "text/plain"});
                response.end(err + "\n");
                return;
            }

            response.writeHead(200);
            response.end(file, "binary");
        });
    });
}).listen(8080);

console.log("Server running at http://localhost:8080/");
```

Yes, this is about as mundane as you can get. Well, that is, until you realize that you’ve actually written a file server in about 20 lines of code. The output you see — the actual code of the script you wrote — isn’t canned in the script itself. It’s being served from the file system. Throw an image into the same directory, and simply add the name of the image to the end of your URL, like *http://localhost:8080/my_image.png*:



Node happily serves this binary image up. That’s pretty remarkable, when you again refer to the brevity of the code. On top of that, how hard would it be if you wanted to write *your own* server code in JavaScript? Not only that, but suppose you wanted to write that code to handle multiple requests? (That’s a hint; open up four, five, or 10 browsers and hit the server.) The beauty of Node is that you *can* write entirely simple and mundane JavaScript to get these results.

A quick line-by-line primer

There's a lot more to talk about around Node than in the actual code that runs a server. Still, it's worth taking a blisteringly fast cruise through *NodeFileServer.js* before moving on. Take another look at the code:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, "127.0.0.1");
console.log('Server running at http://127.0.0.1:1337/');
```

First, you have a call to a function called `require()`. The use of `require()` has been a long-standing request by programmers. You can actually find this mentioned in some of the discussions on JavaScript modularity, as well as germane to CommonJS, and a pretty cool implementation by O'Reilly author David Flanagan from 2009. In other words, `require()` may be new to you, but it isn't an untested, careless piece of Node code. It's core to using modular JavaScript, and something of which Node takes heavy advantage.

Then, the resulting `http` variable is used to create a server. That server is handed a function block to run when it's contacted. This particular function ignores the request completely and just writes out a response, in `text/plain`, saying simply "Hello World\n". Pretty straightforward stuff.

In fact, this lays out the standard pattern for Node usage:

1. Define the type of interaction and get a variable for working with that interaction (via `require()`).
2. Create a new server (via `createServer()`).
3. Hand that server a function for handling requests.
   - The request handling function should include a request ...
   - ... and a response.
4. Tell the server to start handling requests on a specific port and IP (via `listen`).

Lost in translation

Despite the ease with which you *can* get a server coded in JavaScript (regardless of whether the actual code-running facility is C or anything else) still begs the question: *Should* you write a server in JavaScript? To really get a handle on the answer to this question, consider a pretty typical use case.

The JSON round trip

You've got a typical web application, HTML front-end with CSS styling, and JavaScript for the validation and communication with a server. And because you're up on the interactive web, you're using Ajax and not relying purely on a form's POST to get data to and from the server. If this is you, then you're probably comfortable with JSON, too, as that's the almost de facto means of sending data across the wire these days.

So you've got an Ajax request that says, for example, “give me more information about some particular guitar on an online auction site.” That request gets thrown across the network to a PHP program running on a server somewhere. The PHP server has to send a lot of information back to the JavaScript requestor, and it's got to send that information in some format that JavaScript can unpack. So the return information is bundled up into an array, which can then be converted to JSON, sort of like this:

```
$itemGuitar = array(
  'id' => 'itemGuitar',
  'description' => 'Pete Townshend once played this guitar while
his own axe ' .
                    was in the shop having bits of drumkit
removed from it.',
  'price' => 5695.99,
  'urls' => array('http://www.thewho.com',
'http://en.wikipedia.com/wiki/Pete_Townshend')
);

$output = json_encode($itemGuitar);
print($output);
```

Back on the client, the JavaScript gets this chunk of information, which has changed slightly because of JSON and transmission. The client basically gets something like this:

```
{
  "id": "itemGuitar",
  "description": "Pete Townshend once played this guitar...",
  "price": 5695.99,
  "urls": ["http://www.thewho.com",
"http://en.wikipedia.com/wiki/Pete_Townshend"]
}
```

This is pretty standard fare. Then, it's easy to convert this text "thing" into an object in JavaScript. You just call `eval()`, like this:

```
var itemDetails = eval('(' + jsonDataString + ')');
```

The result is a nice JavaScript object with properties that match up to the JSON array-like structure. Of course, since the `jsonDataString` usually is returned from a server, you're more likely to see code like this:

```
var itemDetails = eval('(' + request.responseText + ')');
```

This is the typical JSON round trip. But there are problems here ... big, big problems.

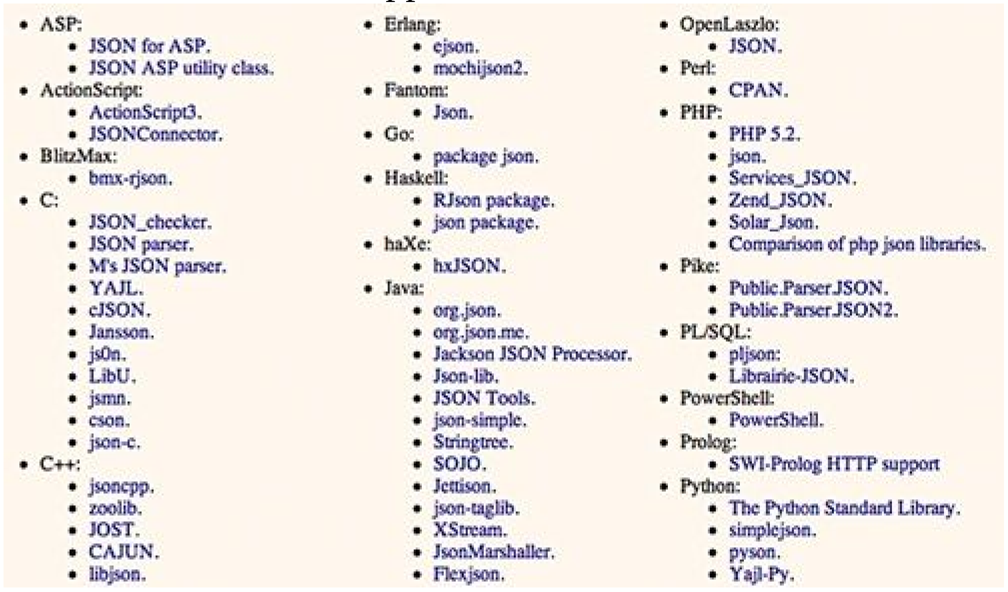
Subtlety and nuance destroy solid code

First, there's a major problem in that this sort of code relies heavily on a translator. In this case, the translator is the JSON interpreter and related code, and there are in fact *two* dependencies: a JSON interpreter for Java in the form of what `eval()` does with the response text, and the JSON interpreter for PHP. As of PHP 5.2.0, that interpreter is included with PHP, but it's still essentially an external dependency, separate from the core of PHP.

Now, this isn't a rant about translation itself. There's nothing to suggest that there are problems in taking, say, an "l" and turning it into an "i", or something that's item 1 in an array and reporting it as being item 2 in an array. There's a lot of testing that occurs before JSON tools are ever released to ensure that what gets reported is correct, and accurate round tripping from a client to a server and back again are possible. Lots and lots and lots of testing is involved ...

And that is in fact a problem.

The dependency of JavaScript and PHP (and C and Lisp and Clojure and Eiffel and ... well, see the figure below for all the JSON toolkits floating around for a ton of different languages) on a toolkit is a huge issue. In other words, the problem isn’t the transla*tion* but the transla*tor*. While programming languages evolve slowly, the uses to which these languages are applied is growing quickly. The result is that JSON is being put to use in areas of complexity that simply didn’t exists or went untouched even a few months ago. And with each new iteration — each new depth of recursion and combination of data types — it’s possible that an area is discovered that the translator doesn’t support.



- ASP:
  - JSON for ASP.
  - JSON ASP utility class.
- ActionScript:
  - ActionScript3.
  - JSONConnector.
- BlitzMax:
  - bmx-rjson.
- C:
  - JSON_checker.
  - JSON parser.
  - M's JSON parser.
  - YAJL.
  - cJSON.
  - Jansson.
  - js0n.
  - LibU.
  - jsmn.
  - cson.
  - json-c.
- C++:
  - jsoncpp.
  - zoolib.
  - JOST.
  - CAJUN.
  - libjson.
- Erlang:
  - ejson.
  - mochijson2.
- Fantom:
  - Json.
- Go:
  - package json.
- Haskell:
  - RJson package.
  - json package.
- haXe:
  - hxJSON.
- Java:
  - org.json.
  - org.json.me.
  - Jackson JSON Processor.
  - Json-lib.
  - JSON Tools.
  - json-simple.
  - Stringtree.
  - SOJO.
  - Jettison.
  - json-taglib.
  - XStream.
  - JsonMarshaller.
  - Flexjson.
- OpenLaszlo:
  - JSON.
- Perl:
  - CPAN.
- PHP:
  - PHP 5.2.
  - json.
  - Services_JSON.
  - Zend_JSON.
  - Solar_Json.
  - Comparison of php json libraries.
- Pike:
  - Public.Parser.JSON.
  - Public.Parser.JSON2.
- PL/SQL:
  - pljson:
  - Librairie-JSON.
- PowerShell:
  - PowerShell.
- Prolog:
  - SWI-Prolog HTTP support
- Python:
  - The Python Standard Library.
  - simplejson.
  - pyson.
  - Yajl-Py.

A selection of JSON toolkits.

That’s not in itself bad. In fact, it argues for the popularity of JSON that it’s constantly put to new use. But with the “new” comes the “does it support the new?” So JSON has to evolve from time to time, and that means testing, and retesting, and release on tons of platforms. You, the programmer, may have to rearrange your data; or wait on a release to support your needs; or hack at JSON yourself. Again, many of these are just the so-called costs of programming.

But imagine you could ditch the translation — and therefore the translator — altogether. Imagine you could write, not JSON round tripping, but JavaScript end to end.

That’s the promise of Node. All the text you’ve just read — about PHP including JSON in 5.2.0 but not before, about arrays becoming objects, about data being configured in new ways and requiring new things from JSON — it all goes away when you have JavaScript sending data *and* receiving and responding to that data.

eval() in JavaScript is (Potentially) the Devil

As if that's not enough reason to look seriously at Node, there's the pesky issue of running `eval()` on a string. It's long been accepted that `eval()` is dangerous stuff. It runs code that you can only see as textual data; it's the equivalent of that "Run Your SQL by typing it in here" unvalidated text box, open to SQL injection and malicious intent. It's quite possible that every time `eval()` is passed in a string, a puppy somewhere in the Midwest shivers and a mom on the Eastern Seaboard stubs her toe and curses. It's that precarious. There's plenty to read about online, and it's not worth going into in detail here. Just Google "eval JavaScript evil" or "eval JavaScript injection" to get a good taste of the issues.

Still, Node without any context doesn't allow you to avoid `eval()`, so there are potentially still shivering puppies out there. However, Node used *as it's intended* absolutely gets you around the typical `eval()` problems. Node is often called *evented JavaScript* or *evented I/O*, and that little word — "evented" — is hugely important. But to get a hold of what evented really means, and why it gets you out of the dangers of `eval()`, you've got to understand not just how JSON is typically round tripped in applications, but how the very structure of applications on the web are typically architected.

Today’s web is a big-event web

Typical web forms are “big-event” submitters. In other words, lots of data entry and selection happens — a user fills out text boxes, selects choices from combo boxes, selects items from a list, and so on — and then all of that information is submitted to a server. There’s a single “big event” from the programming perspective: the submission of all that form data, usually through a POST. That’s pretty much how the web operated, pre-Ajax.

Sending lots of data at one time

With Ajax, there is a little more of what’s called *evented* programming. There are more events that trigger interaction with the server. The classic case is the entry of a zip code, and then a resulting call to the server to get a city and state. With Ajax and the XmlHttpRequest, tons of data didn’t have to be gobbed up and thrown to the server all at once. However, that doesn’t change the reality that the web is still *mostly* a big-event place. Ajax is used far more often to achieve interesting visuals, do quick validations, and submit forms without leaving a page than it is to create truly evented web pages. So even though a form isn’t submitting a big gob of information with a POST, an Ajax request is doing the same thing.

Honestly, that’s only partly the fault of less-than-creative Ajax programmers. Every time you send off a request — no matter how small — there’s a lot of network traffic going on. A server has to respond to that request, usually with a new process in its own thread. So if you really move to an evented model, where you might have 10 or 15 individual micro-requests going from a single page to a server, you’re going to have 10 or 15 threads (maybe less, depending on how threads are pooled and how quickly they’re reclaimed on the server) firing up. Now multiply that by 1,000 or 10,000 or 1,000,000 copies of a given page floating around ... and you could have chaos. Network slowdown. System crashes.

The result is that, in most cases, the Web *needs* to be, at a minimum, a medium-event place. The result of this concession is that server-side programs aren’t sending back tiny responses to very small and focused requests. They’re sending back multiple bits of data, and that requires JSON, and then you’re back to the `eval()` problem. The problem is `eval()`, sure, but the problem is also — from a certain perspective, at least — the nature of the web and threading and HTTP traffic between a web page and a server-side program responding to that request.

(Some of you more advanced JavaScript folks are screaming at this point, because you know better than to use `eval()`. Instead, you're using something like `JSON.parse()` instead of `eval()`. And there are also some compelling arguments for careful usage of `eval()`. These are things worth screaming about. Still, just see how many questions there are surrounding `eval()` on sites like Stack Overflow and you'll realize that most folks don't use `eval()` correctly or safely. It's a problem, because there are lots of intermediate programmers who just aren't aware of the issues around `eval()`.)

Sending a little data at all times

Node brings a different approach to the party: it seeks to move you and your web applications to an evented model, or if you like, a "small event" model. In other words, instead of sending a few requests with lots of data, you should be sending tons of requests, on lots of events, with tiny bits of data, or requests that need a response with only a tiny bit of data. In some cases, you have to almost recall your GUI programming. (All the Java Swing folks can finally use their pent-up GUI knowledge.) So a user enters their first and last name, and while they're moving to the next box, a request is already requesting validation of just that name against existing names. The same is true for zip codes, and addresses, and phone numbers. There's a constant stream of requesting and responding happening, tied to almost every conceivable event on a page.

So what's the difference? Why is this possible with Node, and aren't the same issues around threading existent here? Well, no, they're not. Node's own site explains their philosophy the best:

> Node's goal is to provide an easy way to build scalable network programs. In the "hello world" web server example ... many client connections can be handled concurrently. Node tells the operating system (through epoll, kqueue, /dev/poll, or select) that it should be notified when a new connection is made, and then it goes to sleep. If someone new connects, then it executes the callback. Each connection is only a small heap allocation.

Node has no blocks, no threads competing for the same resource (Node is happy to just let things happen however they happen), nothing that has to start up upon request. Node just sits around waiting (quite literally; unused Node responders are sleeping). When a request comes in, it's handled. This results in very fast code, without uber-programmers writing the server-side behavior.

Yes, chaos can ensue

It’s worth pointing out that this model does allow all the problems that any non-blocking system allows to come into play: one process (not thread) writing to a data store while another one grabs just-invalidated data; intrusions into what amounts to a transaction; and so on. But realize that the majority of event-based programming on a web form is *read-only*! How often are you actually modifying data in a micro-request? Very rarely. Instead, there’s a constant validation, data lookup, and querying going on. In these cases, it’s better to just fire away with the requests. The database itself may add some locking, but in general, good databases will do this much more efficiently than server-side code, anyway; and they’ll certainly handle things better than an operating system will spin up and down threads for a generic, “a web response came in” process.

Additionally, Node does have plans to allow for process forking, and the HTML5 Web Workers API is the engine that will probably make this feature go. Still, if you move to an evented model for your web application, you’ll probably run into an issue where you might *want* threading in less than one out of 100 situations. Still, the changes are best in how you think about your web applications, and how often you send and receive data from a server, rather than in how Node works.

In the right place at the right time

There's another web pattern at work here, and it's probably far more important than whether you use Node or not, and how evented your web applications are. It's simply this: use different solutions for different problems. Even better, use the right solution for a particular problem, regardless of whether that's the solution you've been using for all your other problems.

The inertia of familiarity

There's a certain inertia in not just web design, but all of programming. That inertia can be stated axiomatically like this: the more you learn, use, and become good at a certain approach or technique or language, the more likely you are to use that approach/technique/language widely. It's one of those principles that sounds good until you dig deeply. Yes, it's good to learn a language or toolkit well, and to employ it widely. But this inertia often causes you to use a tool *because* you know it, rather than because it's the *right* tool.

Look at Ajax, something already discussed. Initially, Ajax provided a solid approach to sending quick requests, without form submissions, to a server. Now it's become a drop-in replacement for *all* form submissions. That's taking a technology, learning it, applying it, and then eventually *over*-applying it. There's still a solid place for form submissions — when a form needs to be submitted! As simple as it sounds, there are tends of thousands of web applications submitting forms with Ajax, just because the lead web developer is up on Ajax.

In the same vein, it's possible to get excited about Node — probably because you buy into all the splendid and wise observations you've been reading — and then use it everywhere. Suddenly, you're replacing all your PHP and Perl back-ends with Node. The result? A mess. In fact, you'll be forced to have several web forms do just what Node *isn't* meant for: submit big chunks of data to JavaScript on the server via Node, and force that JavaScript to either send back a chunk of JSON that's got to be parsed or `eval()`ed, or send back a full-blown HTML page or an HTTP redirect.

But that's simply not what Node is best at. It's great at micro-requests; at evented I/O. Use Node for quick communication between a web page and a server. Use form submissions to send big chunks of data to the server. Use PHP and Perl to do heavy database lifting and generate dynamic HTML pages. Use Node to provide a means for server-side JavaScript to run and handle small requests. Throw in Rails and Spring and servlets and whatever else you need. But make your decisions based upon the problem you’re solving, rather than what you happen to know best at the time.

Node’s promise of simplicity

There’s one last note worth making. When you take this broad approach to programming, you’ll often find that you’re not having to go as deeply into each toolkit, API, and framework you use. By using your tools for what they’re best at, you don’t need to be able to staple with hammers or measure with drills. Using tools for their intended purpose typically means you use the core capabilities more. So while you’re creating generalists — programmers that know lots of things — you are also reducing the need for specialists — programmers that know one or two things *really*, *really* well. Of course, every pointy-haired boss also realizes that those specialists are *really*, *really* expensive and hard to find.

Learning Node might take a little effort, but it’s going to pay off. Why? Because you’re afforded solutions to your web application problems that require only JavaScript to solve. That means your existing JavaScript expertise comes into play. And when you do need to use PHP or Perl — because it’s the right solution for a particular problem — you don’t need a PHP or Perl guru. You need to know the basics, and those needs can be expanded when the problem requires expansion. Stretching comes at the behest of new problems, rather than stretching poor solutions thinly.

Your biggest challenge is the continual move to a web that is made up of smaller pieces, talking more often, and the combination of what can seem like a dizzying array of technologies. However, taking the core features of 100 technologies is always going to serve you better than taking 100% of one technology and trying to solve 100 problems. Node and evented I/O isn’t a solution to every problem, but it sure is a solution to some important problems.

Related:

- Node: Up and Running (Book)
- The secrets of Node’s success

- Why a JavaScript hater thinks everyone needs to learn JavaScript in the next year
- JavaScript spread to the edges and became permanent in the process
- What is Node.js and what does it do? (Video)

About the Author

Brett McLaughlin has become one of the most well-known authors and programmers in the Java and XML communities. He's worked for Nextel Communications, implementing complex enterprise systems, at Lutris Technologies, actually writing application servers, and most recently at O'Reilly Media, Inc., where he continues to write and edit books that matter. His most recent book, "Java 5.0 Tiger: A Developer's Notebook", was the first book available on the newest version of Java, and his classic Java and XML remains one of the definitive works on using XML technologies in Java.

What is Node.js?

Brett McLaughlin

Editor
Mike Loukides

O’Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472

2013-05-01T15:29:26-07:00

What is Node.js?

Table of Contents