IBM

developerWorks

Node.js for Java developers

Lightweight, event driven I/O for your web apps

Andrew Glover
CTO
App47

29 November 2011

Node.js untangles concurrency by replacing the Java platform's standard, multithreaded approach with single-threaded, event-driven I/O. In this article, Andrew Glover introduces Node.js and explains why its event-driven concurrency has sparked so much interest, even among die-hard Java developers. He then shows you how to leverage Node's Express framework, Mongolian DeadBeef, and MongoDB to build a concurrent, scalable, and persistent web application.

JavaScript has over the past few years emerged as an under-sung hero of web application development. This recognition has been often met by surprise from software developers accustomed to dismissing JavaScript as a "toy language." While there are more popular languages (in the sense that developers clamor to announce their allegiance to them), JavaScript's unique status as *the* standard, browser-neutral scripting language has lent it staying power. For client-side web development, it is possibly the most widely used language in the world.

JavaScript for Java developers

JavaScript is an important tool for the modern Java developer, and it's not hard to pick up. Follow along as Andrew Glover introduces the syntax you need to build first-class web applications, including JavaScript variables, types, functions, and classes.

JavaScript also has a place in server-side scripting, and that niche is growing. While there have been attempts at server-side JavaScript in the past, none has seemingly captured as much excitement as Node.js, or Node.

Designed to assist developers in building scalable network programs, Node is a server-side programming environment that has virtually reinvented JavaScript for a whole new generation of developers. For many Java developers, Node's biggest pull is its fresh approach to software concurrency. While the Java platform continues to evolve in its approach to concurrency (with greatly anticipated improvements in Java 7 and Java 8), the fact is that Node meets a need, and it does so in the more lightweight fashion that many Java developers have recently embraced.

Like client-side scripting with JavaScript, server-side scripting in the Node environment is so good because it just works, and it's working in an area where many Java developers are currently challenged.

In this article, I introduce you to the server-side scripting revolution that is Node. We'll start with an architectural overview of what makes Node special, and then I'll show you how to quickly build a scalable web app that leverages MongoDB for data persistence. You'll see first-hand how fun Node is, and how quickly you can use it to assemble a working web application.

Node's event-driven concurrency

Node is a scalable, event-driven I/O environment built on top of Google's V8 JavaScript engine. Google V8 actually compiles JavaScript into native machine code prior to execution, resulting in extremely fast runtime performance — something not typically associated with JavaScript. As such, Node enables you to rapidly build network apps that are lightning fast and highly concurrent.

Event-driven I/O might sound foreign to a Java developer's ears, but it isn't all that new. Rather than the multithreaded programming model we're used to on the Java platform, Node's approach to concurrency is single-threaded, with the additional kick of an *event loop*. This Node construct enables non-blocking or asynchronous I/O. In Node, invocations that typically would block, such as waiting for database query results, don't. Rather than wait around for expensive I/O activities to complete, a Node application issues a callback. When a resource is returned, the attached callback is invoked asynchronously.

Why choose Node.js?

The Java platform's approach to concurrency helped to establish its leading role in enterprise development, and that isn't likely to change. Frameworks like Netty (and Gretty; see Resources) and core libraries like NIO, as well as `java.util.concurrent`, have made the JVM a top choice for concurrency. What's special about Node is that it is a modern development environment specifically designed to resolve the challenges of concurrent programming. Node's event-driven programming paradigm means that you don't need additional libraries in order to make concurrency work, and that's good news for developers eyeing multicore hardware.

Concurrency *just works* in Node programs. If I wanted to run the previous scenario on the Java platform, I would have my choice of complex and time-consuming approaches — from traditional threads to newer libraries in Java NIO and even the improved and updated `java.util.concurrent` package. While Java concurrency is powerful, it can be difficult to understand — which translates to code! Node's callback mechanism, by comparison, is built into the language; you don't need special constructs like `synchronized` in order to make it work. Node's concurrency model is extremely simple, and that makes it accessible to a wider audience of developers.

Node's JavaScript syntax also saves you keystrokes, big time. With just a little code, you can build a fast, scalable web app that is capable of handling myriad concurrent connections. You can do that on the Java platform, but you'll need more lines of code and a bevy of additional libraries and constructs. And if you're worried about navigating a new programming environment, don't be: Node is easy to pick up if you know some JavaScript, which I'm betting that you do.

Getting started with Node

As I've mentioned, Node is easy to get started with, and you'll find a number of good tutorials online to help you there. In this article (as in my Java developer's introduction to JavaScript) I'm focusing on helping Java developers reap the benefits of Node. Rather than walk you through the standard "Hello, world" web server application, I want to dive straight in to a meaningful app that really does something: think of it as a Foursquare(ish) clone built on Node.

Video demo: Getting started with Node.js

Prefer a different approach to getting started with this handy framework? Watch this video demo, also by Andrew Glover, to learn more about Node.js and what it can do for you. (Read the transcript.)

Installing Node requires that you follow the instructions for your particular platform; if you're using a UNIX-like platform such as OSX, I recommend using Node Version Manager, or NVM, which handles the details of installing an appropriate version of Node. In any case, download and install Node now.

We'll also be using some third-party libraries to build this app, so you'll want to install NPM, which is Node's package manager. NPM allows you to specify versioned dependencies of your project, which can then be downloaded and included in your build path. NPM is in many ways similar to Maven for the Java platform, or Ruby's Bundler.

Node Express

Node is finding a nice home among web developers, both for its ability to handle concurrency and because it was built with web development in mind. One of the most popular third-party Node tools is the lightweight web development framework Express, which we'll be using to develop our app (see Resources to learn more about Express).

Express is loaded with features, including sophisticated routing, dynamic template views (see the Node framework *du jour:* Jade), and content negotiation. Express is also pretty lightweight, with no embedded ORM or similar baggage to weigh it down. In this, Express isn't comparable to Rails, Grails, or any other full-stack web framework.

An easy way to install and leverage Express is to declare it as a dependency via an NPM `package.json` file, shown in Listing 1. This file is similar to Maven's `pom.xml` or Bundler's `Gemfile`, but its format is JSON.

Listing 1. NPM's package.json file

```
{
  "name": "magnus-server",
  "version": "0.0.1",
  "dependencies": {
    "express": "2.4.6"
  }
}
```

In Listing 1, I've given this Node project a name (magnus-server) and a version (0.0.1). I've also declared version 2.4.6 of Express as a dependency. The nice thing about NPM is that it will grab all of Express's transitive dependencies, swiftly loading up any other third-party Node libraries that Express requires.

After you've defined your project's dependencies via `package.json`, you can install the desired packages via the command-line by typing `npm install`. You should see NPM install Express along with dependencies such as *connect*, *mime*, and more.

Writing a network app

We start writing our example application, Magnus Server, by creating a JavaScript file; I've named mine web.js but you can name it whatever you like. Open that file in your favorite editor or IDE; for example, you could use JSDT, the Eclipse JavaScript plugin (see Resources).

In the file, add the code from Listing 2:

Listing 2. Magnus Server: First cut

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

var app = express.createServer(express.logger());

app.put('/', function(req, res) {
  res.contentType('json');
  res.send(JSON.stringify({ status: "success" }));
});

var port = process.env.PORT || 3000;

app.listen(port, function() {
  console.log("Listening on " + port);
});
```

This little code snip does some pretty big things, so I'll start from the top. First, if we want to use a third-party library in Node, we have to use the `require` phrase; in Listing 2, we're *requiring* the Express framework, and also getting a handle to it via the `express` variable. Next, we create an app instance via the `createServer` call, which in turn creates an HTTP server.

We then define an endpoint, via `app.put`. In this case, we're defining an HTTP `PUT` as the required HTTP method listening at the root of the application (`/`). The `put` call has two parameters: the route and a corresponding callback to be executed when that route is invoked. The second parameter is a function that is invoked at runtime when the endpoint `/` is hit. Remember, this callback is what Node means by event-driven or *evented* I/O — the callback will be invoked asynchronously. This endpoint can handle myriad simultaneous requests without the need to manually create threads.

As part of the endpoint definition, we then create the logic to handle a `PUT` to `/`. We'll keep things simple for now by setting the response type to JSON and then sending back a simple JSON document: `({"status":"success"})`. Notice the nifty `stringify` method, which takes a hash and converts it to JSON.

JavaScript and JSON

JSON and JavaScript are practically siblings, and that affinity translates to Node. Parsing JSON within a Node app doesn't require special libraries or constructs; instead, you use logical calls that are similar to object-graphs. In short, Node treats JSON like family, which makes programming JSON-based web applications that much easier.

Next, we create a variable representing the port the app should listen on; we do this by either obtaining the `PORT` environment variable or explicitly setting it to 3000. Finally, we fire up the app by calling the `listen` method. Once again we pass in a callback, which will be invoked once the app is up and running to print a message to the console (in this case, `standard out`).

Give it a try!

This nifty app responds to any `PUT`, so you can run it simply by typing in `node web.js` at the command-line. If you want to further test the app, I recommend downloading WizTools.org's RESTClient. With RESTClient, you can quickly find out whether Magnus Server works by executing an HTTP `PUT` to http://localhost:3000. If all goes well, you should see a JSON response indicating success. (See Resources to learn more about installing and using RESTClient.)

Express wrangles JSON

JavaScript and JSON are tightly related, which makes managing JSON in Express about as simple as it gets. In this section, we'll add a bit more code to the skeleton app in Listing 2, so that we can grab an incoming JSON document and print it to `standard out`. After that, we'll persist the whole thing to a MongoDB instance.

The incoming document will look something like Listing 3 (note that I've omitted location information for the sake of brevity):

Listing 3. Free food at Freddie Fingers!

```
{
 "deal_description":"free food at Freddie Fingers",
 "all_tags":"free,burgers,fries"
}
```

Listing 4 adds functionality to parse the incoming document:

Listing 4. Express parses JSON

```
app.use(express.bodyParser());

app.put('/', function(req, res) {
  var deal = req.body.deal_description;
  var tags = req.body.all_tags;

  console.log("deal is : "  + deal + " and tags are " + tags);

  res.contentType('json');
  res.send(JSON.stringify({ status: "success" }));
});
```

Note that Listing 4 includes a line directing Express to use `bodyParser`. This will enable us to easily (and I do mean *easily*) grab attributes from an incoming JSON document.

Inside the `put` callback is code to obtain the values for an incoming document's `deal_description` and `all_tags` attributes. Note how easily we're able to grab the individual elements of a requested document: In this case, `req.body.deal_description` obtains the value of `deal_description`.

Test it!

You can test this implementation out, too. Just stop your `magnus-server` instance and restart it, then use an HTTP `PUT` to submit a JSON document to your Express app. First, you should see a successful response. Second, Express should log to `standard out` the values you've submitted. With my Freddie Fingers document, I get the output

```
deal is : free food at Freddie Fingers and tags are free, burgers, fries.
```

Persistence with Node

We have a working application that receives a JSON document, parses it, and returns a response. Now we need to add some persistence logic. Because I'm a fan of MongoDB (see Resources), I choose to persist the data via a MongoDB instance. To make things extra easy, we'll leverage the third-party library Mongolian DeadBeef, which we'll use to store the values of incoming JSON documents.

Mongolian DeadBeef is one of quite a few MongoDB libraries available for Node. I choose it because its name amuses me, and also because its mirroring of the native MongoDB driver makes me feel right at home.

By now, you know that the first step to using Mongolian DeadBeef is to update the `package.json` file, as shown in Listing 5:

Listing 5. Adding JSON parsing

```
{
  "name": "magnus-server",
  "version": "0.0.1",
  "dependencies": {
    "express": "2.4.6",
 "mongolian": "0.1.12"
  }
}
```

Because we're hooking up with a MongoDB datastore, we also need to update the project's hard dependencies by running `npm install`. For increased performance of the Mongolian DeadBeef MongoDB driver, we could also install the native C++ bson parser, something NPM could guide us through.

To begin using Mongolian DeadBeef, we add another `require` to the current implementation, then connect to the desired MongoDB instance (as shown in Listing 6). For this example, we'll connect to an instance hosted at MongoHQ, a cloud provider of MongoDB.

Listing 6. Adding Mongolian DeadBeef to magnus-server

```
var mongolian = require("mongolian");
var db = new mongolian("mongo://a_username:a_password@flume.mongohq.com:23034/magnus");
```

Inside the `PUT` callback, we persist the values from the incoming JSON document, as Listing 7 shows:

Listing 7. Adding Mongolian insert logic

```
app.put('/', function(req, res) {
  var deal = req.body.deal_description;
  var tags = req.body.all_tags;

  db.collection("deals").insert({
     deal: deal,
     deal_tags: tags.split(",")
  })

  res.contentType('json');
  res.send(JSON.stringify({ status: "success" }));
});
```

Looking closely, you'll see that the `insert` statement looks identical to an insert inside the MongoDB shell. This is no coincidence — MongoDB's shell uses JavaScript! Consequently, we can easily persist a document that has two fields: `deal` and `deal_tags`. Note how we set `deal_tags` to an array by using the `split` method on the `tags` string.

Can I kick it? (Yes you can!)

If you want to test this out (and who wouldn't?) restart your instance, send along another JSON document, and then proceed to check your `deals` collection inside MongoDB. You should see a JSON document almost identical to the one you sent.

Listing 8. Adding Mongolian insert logic

```
{
  deal:"free food at Freddie Fingers",
  deal_tags: ["free", "burgers", "fries"],
  _id: "4e73ff3a41258b7423000001"
}
```

In conclusion — that's it!

If you think I'm slacking by ending this little introduction to Node.js so soon, then I've got news for you: we're finished! We've only written about 20 lines of code, but they add up to a full-fledged, persistent application — and that's the beauty of Node. It's amazingly simple to write and comprehend, and asynchronous callbacks make it extremely powerful. Once an app is written, it can be deployed to any number of PaaS providers, for maximum scalability.

See the Resources section to learn more about the technologies discussed in this article, including Node.js, MongoDB, and PaaS options such as Google App Engine, Amazon's Elastic Beanstalk, and Heroku.

Resources

Learn

- *Java development 2.0*: Explore the technologies that are redefining the Java development landscape, including: JavaScript for Java developers (April 2011), Kilim (April 2010), RESTClient (November 2009), Gretty (August 2011), and MongoDB (September 2010).
- "What is Node.js?" (Michael Abernethy, developerWorks, May 2011): Learn more about Node, a server-side JavaScript interpreter that has revolutionized common ideas about how a server should work.
- "Use Node.js as a full cloud environment development stack" (Noah Gift and Jeremy Jones, developerWorks, April 2011): Discusses the Node framework and ecosystem, then presents an exercise in using Node to build a chat server.
- Java concurrency knowledge path: A developerWorks resource for software developers who want to master multithreaded programming on the Java platform, as well as alternate approaches that exploit multicore processor hardware.
- Knowledge path: Cloud computing fundamentals: Introduces cloud computing concepts and the service models IaaS, PaaS, and SaaS.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.
- Browse the Java technology bookstore for books on these and other technical topics.

Get products and technologies

- Learn more about the technologies discussed in this article:
  - Node.js
  - Node Package Manager
  - Node Version Manager
  - Node Express web framework
  - Jade: A Node template engine
  - Mongolian DeadBeef
  - MongDB

Discuss

- Get involved in the My developerWorks community: Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Andrew Glover

Andrew Glover is a developer, author, speaker, and entrepreneur with a passion for behavior-driven development, Continuous Integration, and Agile software development. He is the founder of the easyb Behavior-Driven Development (BDD) framework and is the co-author of three books: Continuous Integration, Groovy in Action, and Java Testing Patterns. You can keep up with him at his blog and by following him on Twitter.