Lecture 5: Node.js and Server-Side JavaScript

Nick Hynes

January 23, 2014

Topics

1 Why Node.js?

Node.js is a web server, a program that runs on a computer, accepts incoming HTTP requests, for resources, and returns the requested resources; this generally occurs over the Internet. There are a great many web server implementations, but Node.js stands out for its real-time and push capabilities that stem from its asynchronous, event-based architecture. Of course, it is also written mostly in JavaScript, which is always nice, especially since this happens to be a course on JavaScript.

While Node.js can be used for standard web server tasks like static file serving (use Nginx) and creating dynamic pages from templates, it really shines when used for making high-performance, real-time applications like collaborative editing and gaming. The key to Node.js's speed is that it's *non-blocking*. What better way is there to explain this than an example? The following example will be of an application that creates new chat rooms.

Example 1 – Building non-blocks

```
/* Blocking/synchronous web servers (pseudo-code) */
roomName = request.roomName
roomInfo = request.roomInfo
if roomExists(roomName): //roomExists checks the rooms DB
  makeRoom(roomName, roomInfo) //writes to the DB
  return "{\"room\": \"+ roomName +\"}"
else:
  return "{\"error\": \"room exists\"}"


/* Node.js: non-blocking/asynchronous web server */
var roomName = request.roomName;
roomExists(roomName, function(exists) {
  if(exists) {
    response.send({ error: "room exists" });
  } else {
    makeRoom(roomName, function(room) {
      room.roomInfo = request.roomInfo;
    });
    //response is sent before the room is fully created
    response.send({ room: roomName });
  }
});
```

While, at first glance, the two snippets may look like they do the same thing, the second is subtly different. For the purposes of demonstration, assume that `roomExists` and `makeRoom` are functions that take a long time to complete since they access the rooms database, which requires querying another program for a value in memory and, in the worst case scenario, a read from disk. Whereas in the first snippet, the response is not returned until `makeRoom` is done, in the second, the response is returned as soon as the room is determined to not exist while `makeRoom` continues in the background. This kind of functionality can be duplicated using, for example, a Java server by running `makeRoom` as a new thread. However, this new thread waits around, consuming resources, until the DB transaction has completed. Node.js, on the other hand, simply sits quietly until it is notified of the transaction's completion, at which point it runs the callback function that completes the room's initialization.

It shouldn't take much convincing to become a fan of non-blocking IO.

2 How to Node

2.1 Installation

Just navigate to `http://nodejs.org` and click on the prominently located "Install" button. You'll be prompted to download the Node.js bundle (Node.js, supporting libraries, and the Chrome V8 JS engine) for your platform.

Linux

Use your package manager, or move the files to the correct places.

Windows

Run the installer and follow its instructions.

Macintosh

Run the installer.

2.2 Installing modules using NPM

When you installed Node.js, you also installed a handy tool that, like python's pip and EasyInstall, allows you to download modules. Node modules are similar to client-side node libraries, but since they have the ability to access the filesystem, they can include and run native programs. You can install and uninstall modules by running `npm install/uninstall package-name` on the command line. You can also find a great variety of modules, from database connectors to websocket APIs, by searching "node module *whatever you're trying to do*".

2.3 A simple server

Example 2 – Ping pong

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

//Makes a 4 second long page loop
http.createServer(function (req, res) {
  if(req.url === "/ping") {
    res.writeHead(200, {"Refresh": "2; url=/pong"});
    res.end("Ping"); //Sends the text "Ping"
  } else if(req.url === "/pong") {
    res.writeHead(200, {"Refresh": "2; url=/ping"});
    res.write("Pong"); //Writes "Pong"
    res.end(); //Sends the response
  } else {
    res.writeHead(301, {"Location": "/ping"});
    res.end();
  }
}).listen(4242, "127.0.0.1");
console.log("Navigate to http://127.0.0.1:4242/");
```

I got tired of writing "Hello, world" examples, so, instead, here's an example of a server that sends a page that redirects to another page, that redirects to the first page.

The Node.js dialect of JavaScript is slightly different from what you've already seen, but the same principles still apply. One of the first things you'll notice is the existence of the global `require` function. This function is used to include modules and other JS files. In this case, the module is the built-in HTTP library. This provides the `createServer` function that creates an object that can be used as an HTTP server. Its single argument is the request event handler function, or the function that gets called when the server receives a request. We then make the new HTTP server `listen` listen for requests on port 4242 of all network interfaces.

You can think of a port as a place where a server can build docks for incoming requests; a server is given exclusive building rights, so no other server can build on (bind) the same port. HTTP uses port 80 (443 for HTTPS), but you may have to run `node` as superuser/Administrator to be able to bind this port. Only the government can build on the important ports with numbers less than 1023, to continue the metaphor.

A network interface is a channel by which the machine can access the network. Most OSes have a network stack that includes an interface known as *localhost*. This interface, also known as the loopback interface, refers to the local machine. Localhost has the IP address 127.0.0.1.

The HTTP module is great and all, but it's relatively low-level and, out of the box, doesn't provide anything but the basic functionality. For this, we'll use a module that provides a framework for making web applications. There are a number of these available that serve different purposes, but we'll be focusing on Express since it's currently the most popular and has many useful high-level functions.

3 The Express track to web apps

Express is a popular Node.js web application framework that builds on the Connect middleware (plug-in) framework that builds on the built-in HTTP module. It's analogous to the Rails in Ruby on Rails and Django and can be used to dramatically increase development speed (it is named Express, after all).

3.1 "Installing" Express

To install Express, simply navigate to your project's folder and type `npm install express`. This will create a folder named "node_modules" that contains the newly-downloaded Express module. You can now use Express in your project by using `require("express");`. If you make a project in a different directory/folder, you will have to install Express again in that directory.

Example 3 – GET me a variable

```
var express = require("express");
var app = express(); //Creates a new Express app

//These includes are run when a request is received
//These includes are known as "middleware"
var __dirname = "/home/nhynes"
app.configure(function () {
  app.use(express.static(__dirname + '/static'));
  //serves resources not matched by a route
});

app.get("/", function(req, res) {
  var name = req.param("name"); //like /?name=Nick
  res.send("Hello " + name); //headers are auto-set
});

app.listen(4242); //starts the server
```

This example is a bit closer to the ubiquitous "Hello world." However, instead of greeting the world, the server sends a personalized message if the user enters a name as the `name` GET variable (e.g. localhost:4242/?name=Geronimo). What's interesting here is that we configure the Express app to use static directoroy serving "middleware" that tries to serves a requested file from disk if the user requests a file that is not the document root. To this end, we create a route that serves the document root by using the `app.get` function.

```
//In general,
app.VERB("route", function(request, response) { });
//if the HTTP method is VERB and the request URL
//matches route, then the function will be called
```

3.2 Routing

When making a REST API as part of your client-server interaction model, you'll likely want to set up descriptive URLs, or routes, that specify what kind of dynamic object the user is requesting. For example, the resources /hat/baseball and /pants/pantaloon would be served by the `/:clothingArticle/:type` route. Here's how you'd use it with Express.

Example 4 – Your very own fashion expert

```
var express = require("express");
var app = express(); //Creates a new Express app

app.get("/:clothingArticle/:type", function(req, res) {
  var article = req.params.clothingArticle;
  var articleType = req.params.type;
  res.send("Your " + articleType + " " + article + " " +
           "looks ridiculous.");
});

app.listen(4242); //starts the server
```

Now, whenever I navigate to localhost:4242/mustache/fancy, I'll be told that "[My] fancy mustache looks ridiculous." So much for my fancy mustache. We're able to get the routing information by accessing the properties of the `req.params` object, as opposed to by calling the `req.param()` function.

This is how routing happens. Really simple! It will help, however, to move the callback functions for your routes into separate files to prevent your app from becoming a monolithic monstrosity that is borderline unmaintainable.

3.3 Accepting form data

Example 5 – Forming a form of form processing

```
var express = require("express");
var app = express(); //Creates a new Express app

var clothes = {};

app.post("/:clothingArticle", function(req, res) {
  var article = req.params.clothingArticle;
  if(clothes[article] === undefined) {
    clothes[article] = [];
  }

  var articleType = req.param("type");
  var clothes[article].append(articleType);

  res.status(200).end(); //Shorthand way to respond
});

app.listen(4242); //starts the server
```

You could imagine this code being run when a user submits a form for making a new type of clothing that makes an AJAX POST request to the given route.

Just for entertainment, this is what it'd look like on the client:

Example 6 – Clothes by POST

```
//This is called when a form containing an input for
//article of clothing and the article type is submitted
function makeClothing() {
  var article = $("#newClothingArticle").val();
  var articleType = $("#newClothingArticleType").val();

  $.post("/"+article, "type="+articleType)
    .done(function() {
      console.log("Enjoy your new clothes!");
    })
    .fail(function() {
      console.log("Unable to make your clothing.
                   This is probably for the best.");
    });
}
```

3.4 A simple "REST API"

Let's put this GET and POST action together and see what kind of super-cool web application we get!

Example 7 – Putting it all together

```
var app = require("express")(); //Make a new Express app

var clothes = {};

app.get("/:clothingArticle", function(req, res) {
  //return a JSON Array of current types of this article
  var article = req.params.clothingArticle || [];
  res.send(clothes[article]);
});

app.get("/:clothingArticle/:type", function(req, res) {
  //Given both article and type, print a silly message
  var article = req.params.clothingArticle;
  var type = req.params.type;

  if(clothes[article] === undefined) {
    res.status(404).end("No such article of clothing");
  } else if(clothes[article].indexOf(type) === -1) {
    res.status(404).end("No such type of "+article);
  } else {
    res.send("Your " + type + " " + article +
           " looks ridiculous.");
  }
});

app.post("/:clothingArticle", function(req, res) {
  var article = req.params.clothingArticle;
  if(clothes[article] === undefined) {
    clothes[article] = [];
  }

  clothes[article].append(req.param("type"));
  res.status(200).end(); //Shorthand way to respond
});

app.listen(4242); //starts the server
```

And there you have it, a high performance server-side application that allows you to create and access lists of clothing. Although this may not look like something that's going to put any fashion retails out of business, it contains all the basics of a HTTP REST API. Most of the rest (no pun intended) of the server functionality will involve something data-related like querying and modifying data in a database or broadcasting data to clients using WebSockets.

At this point, you should feel confident that you have all of the tools necessary to go off into the world and start making web applications! Web applications written entirely in JavaScript, at that!