Alan Storm
the professional weblog;

Modern Javascript for PHP Developers

Frustrated by Magento? Then you'll love Commerce Bug, the must have debugging extension for anyone using Magento. Whether you're just starting out or you're a seasoned pro, Commerce Bug will save you and your team hours everyday. Grab a copy and start working **with** Magento instead of against it.

This entry is part 1 of 4 in the series Modern Javascript for PHP Developers. Later posts include Express and NPM for PHP Developers, Client Side Javascript, Modules, and Webpack, and Build Watchers, and NPM as a Build Tool.

It's getting harder and harder to ignore the modern javascript world. Even if you plan on staying safely ensconced in your non-javascript platform of choice, working with a web browser means javascript based or enhanced UIs. More and more these platforms are adopting a modern javascript tool chain. The days of *include this javascript file as a `<script/>` tag* are coming to an end.

In my new Modern Javascript for PHP Developer series we'll explore the world of javascript development. The main audience for this series is PHP developers looking to get up to speed quickly on modern javascript projects. However, we'll try to keep things simple and generic enough that all you'll need is a basic grasp of development. The only prerequisites are

1. You've installed NodeJS on your computer via an installer or the Node Version Manager
2. You know how to open a command line terminal and run commands
3. You have `git` installed on your computer

You can test all the prerequisites by running the following commands from your terminal application.

```
$ node -v
v6.4.0

$ npm -v
3.10.6

$ git --version
git version 2.8.1
```

If you get similar output, you're ready to go (Most NodeJS installers will include both the `node` and `npm` command line programs).

Today we're going to show you how to write a program in javascript that responds to an HTTP request. In other words: Server side javascript.

Server Side Javascript

If you're a PHP developer, you probably think of web programming infrastructure/architecture in a very specific way. Namely

- A user requests a URL from a web server
- The web server maps that URL to a PHP file
- The PHP file serves as the program's main entry point

If you’ve used an MVC framework in the past 10 years you might replace *a PHP file* with `index.php` and include the extra step of *index.php bootstraps the framework and routes execution to the controller action entry point for this URL*.

Server side javascript has a different infrastructure/architecture model. It’s still web development — that is, a javascript application still responds to HTTP and HTTPS requests. However, the way your program runs is a bit different and this, in turn, will have consequences on how your program behaves.

Let’s get started with some good old fashioned hello world programs. First, create the following file with the following contents in a folder on your computer.

```
//File: start.js
console.log("Hello World");
```

Then, run your program with the command line `node` command

```
$ node start.js
Hello World
```

Congratulations! You’ve just written your first line of non-browser javascript. Of course, this code isn’t on the web yet. As a PHP developer, your first thought may be

> Where do I put this file so my web-server can see it?

```
//File: start.js
console.log("Program Started");
var http = require('http');

//Configure our HTTP server to respond with Hello World to all requests.
var server = http.createServer(function (request, response) {
    console.log("Processing Request");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.end("Hello World\n");
});

// Listen on port 8000, IP defaults to 127.0.0.1
server.listen(8000);
console.log("Program Ended");
```

There's a bunch of new stuff in here — before we get to explaining it all, try running your program again.

```
$ node start.js
Program Started
Program Ended
▒
```

You'll notice two things — first, we see both the `Program Started` and `Program Ended` messages. Second, and more importantly, we **do not** see our terminal prompt. Our program is still running, waiting around for something to happen.

```
http://localhost:8000
```

The `localhost` server name means *this computer*, and the `:8000` specifies a different networking port to make the request on/with. If you're not familiar with network ports, they're how a computer keeps track of messages coming in on different protocols and/or from different sorts of applications. If you don't include a port with your URL, almost all browsers will default to `80` for HTTP, and `443` for HTTPS.

If you loaded the page in your browser, you should see a *Hello World* message. If you loaded the page via `curl -i` (the `-i` tells curl to include the HTTP headers) you should see the following

```
$ curl -i http://localhost:8000
HTTP/1.1 200 OK
Content-Type: text/plain
Date: Mon, 08 May 2017 19:40:33 GMT
Connection: keep-alive
Transfer-Encoding: chunked

Hello World
```

Also, if you look back at the terminal where we left our NodeJS program running, you'll see the following

```
$ node start.js
Program Started
```

That is, our program is still running **and** our program has sent a `Processing Request` message to the screen.

We are definitely not in PHP land anymore. You can press `Ctr-C` to end your program.

New Concepts

Looking back at our original program.

```
//File: start.js
console.log("Program Started");
var http = require('http');

//Configure our HTTP server to respond with Hello World to all requests.
var server = http.createServer(function (request, response) {
  console.log("Processing Request");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("Hello World\n");
});

// Listen on port 8000, IP defaults to 127.0.0.1
server.listen(8000);
console.log("Program Ended");
```

There's at least three concepts here that may not be familiar to the average PHP programmer. The `require` statement, the passing-a-function to the `createServer` method,

require

In NodeJS, the `require` statement loads a *code module*. NodeJS uses the CommonJS module standard to organize its internal libraries. The following command

```
//File: start.js
var http = require('http');
```

loads the `http` library/module.

http.createServer

The following code uses the `createServer` method of the `http` module/library

```
//File: start.js
var server = http.createServer(function (request, response) {
  console.log("Processing Request");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("Hello World\n");
});
```

In NodeJS (and server side javascript in general), **your program is a web server**. When you call the `createServer` method, you're creating a web server object. You pass `createServer` a javascript function. When you use a function like this in NodeJS, that function is often called a *callback*. The `createServer` method will call your function when it needs to handle an HTTP request.

Finally, we have the following code

```
//File: start.js
server.listen(8000);
console.log("Program Ended");
```

This tells our program it should not stop running once the main entry point finishes execution. Instead, our program should listen for incoming HTTP requests on port 8000. Remember when we used port `8000` in our URL? This `listen` call is what tells our program to monitor the computer’s network layer for any requests to port `8000` and respond to them.

When you consider this entire sequence, something may seem off. If you consider the initial output of our console after running the program

```
$ node start.js
Program Started
Program Ended
▒
```

you’ll recall the program output `Program Ended`. However, in our code, we called `listen` **before** we called `console.log("Program Ended")`. It’s not unreasonable for you to think the call to `server.listen` should have prevented the final line of our program from running.

This is one of NodeJS’s most famous features — non-blocking, asynchronous code. **Culturally** speaking, a node method should never stop and wait for something to finish. This may seem impossible for something like a *wait and listen* method — javascript still implemented in a NodeJS library.

All of that is another topic for another day, but worth mentioning to clear up why our program reached the `console.log('Program Ended');` line.

Responding to a Request

Now that we understand the execution of our main program, we need to dive into the execution of our callback function. That is, the function that responds to HTTP requests. Here's the response callback

```
//File: start.js
function (request, response) {
  console.log("Processing Request");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("Hello World\n");
};
```

When a user loads the `http://localhost:8000` URL in a browser, the system will call this function. The system calls this function because it's the one we passed into `http.createServer`. The object in the `request` variable will contain information about the request (the URL requested, `GET` variables, `POST` fields, cookies, etc.).

The object in the `response` variable has methods that allow us to tell our program what to send back to the user. The `writeHead` method lets us set a `Content-Type` for the response. The `response.end` variable lets us both 1. End our response to this particular request, and 2. Set a string as the HTTP body. If we'd wanted to send back some HTML, we would have done something like this

```
    //send as text/html so the browser knows how to render
    response.writeHead(200, {"Content-Type": "text/html"});

    //include HTML tags in our response
    response.end("<h1>Hello World</h1>\n");
```

Global State

One of NodeJS's biggest differences from PHP is that our program stays resident in memory. If you understand how folks deploy PHP you know that some sort of PHP process is always resident in memory (via MOD_PHP, Fast CGI, PHP-FPM, etc), but **your program** runs once, and then stops.

With server side javascript your program will stay running. This has some performance and scaling benefits, but also has some non-obvious consequences. For example, change your `start.js` to the following

```
//File: start.js
console.log("Program Started");
var http = require('http');

var counter = 0;
//Configure our HTTP server to respond with Hello World to all requests.
var server = http.createServer(function (request, response) {
    console.log("Processing Request");
    response.writeHead(200, {"Content-Type": "text/plain"});
    counter++;
```

```
// Listen on port 8000, IP defaults to 127.0.0.1
server.listen(8000);
console.log("Program Ended");
```

Here we've changed our program slightly. We've defined a `counter` variable

```
var counter = 0;
```

and then, in our request handling callback, we increment the counter and include its value in our response.

```
counter++;
response.end("Hello World: " + counter + "\n");
```

If you start your program again and start requesting our localhost URL over and over again

```
$ curl http://localhost:8000
Hello World: 1
$ curl http://localhost:8000
Hello World: 2
$ curl http://localhost:8000
Hello World: 3
$ curl http://localhost:8000
Hello World: 4
$ curl http://localhost:8000
```

```
    Hello World: 6
```

you'll see our program *remembers* the value of `count` after each request.

This is radically different than PHP's *run once, throw everything away* model. It also makes global state a much more dangerous thing in server side javascript programs, as **any** request from any user might have fiddled with a variable.

Finally, while we won't get into it today, this also has consequences when we deploy our systems to production. We deploy most NodeJS applications with something called a process manager. A process manager can start and stop node processes on a whim. In our above example, this means our counter variable might be reset back to zero at anytime.

If you're used to using little global state hacks in PHP — you'll need to seriously reconsider your approach to programming. Global state in a javascript program can be disastrous.

Wrap Up

Building web application in server side javascript is lot like, as an english speaker, visiting the UK, Australia, Canada, or the United States. People are speaking the same language — but everything seems turned slightly, and some things are just flabbergastingly different.

Today we covered the basic execution mode that **all** NodeJS based programs use. Next time we'll look at our first NodeJS framework, and learn how it can help us organize our NodeJS applications.

Series Navigation

Express and NPM for PHP Developers >>

Alan Storm
the professional weblog;

MODERN JAVASCRIPT

SHARE: