Express and NPM for PHP Developers

This entry is part 2 of 4 in the series Modern Javascript for PHP Developers. Earlier posts include Modern Javascript for PHP Developers. Later posts include Client Side Javascript, Modules, and Webpack, and Build Watchers, and NPM as a Build Tool.

Last time we investigated the basic execution flow of a server side NodeJS program. Our program responded directly to an HTTP request sent by a web-client. Most NodeJS programs aren't written like this — instead, they use a routing framework of some kind to help separate requests by their URLs and other request parameters. Today we'll look at one of the most popular routing frameworks, Express.

This article assumes you have a working installation of NodeJS on your computer. It will also help if you've worked your way through the first article in this series, but things are still simple enough that you should be able to wing it if you haven't.

Routing

Last time we wrote a very simple HTTP server in NodeJS. If you skipped that article, just create a new folder with the following file

```
//File: our-express-project/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 for URL" + request.url);
    response.writeHead(200, {"Content-Type": "text/html"});
    response.end("Hello World \n");
});

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

and then run the program with

```
$ cd our-express-project
$ node start.js
```

This program listens for requests on port `8000`, and we can view its HTTP output by loading the following URL in a web browser or via the command line `curl` program.

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

The program itself will stay persistent in memory and continue to run. NodeJS sends `console.log` calls to the terminal running the program, not to the web browser.

Last time we talked a bit about how this is different from PHP's infrastructure/architecture model. There's one aspect of this we didn't discuss. Try loading a few more URLs

```
http://localhost:8000/hello
http://localhost:8000/world
http://localhost:8000/foo-baz-bar
```

Each of these URLs returns **the same** HTML.

```
$ curl 'http://localhost:8000/hello'
Hello World

$ curl 'http://localhost:8000/world'
Hello World

$ curl 'http://localhost:8000/foo-baz-bar'
Hello World
```

Theo only thing the `http.createServer` callback function does is *respond to a request to the server **from any** URL*. This is different from how we think about things in PHP. In PHP, the web server maps a URL to a PHP file.

NodeJS does give us access to a URL — you can see this by stopping your program (with Ctrl-C), replacing the `http.createServer` block with the following, and then restarting your program

```
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 for URL: " + request.url);
```

```
    response.writeHead(200, {"Content-Type": "text/html"});
    response.end("Hello World \n");
});

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

With the above program running, you'll see the following output in your console when you request a URL.

```
Processing Request for URL: /foo-baz-baz
```

If you're loading your URLs in a web browser, you may see *two* lines in your console.

```
Processing Request for URL: /foo-baz-baz
Processing Request for URL: /favicon.ico
```

The second is your browser automatically requesting a `favicon` in the background. Like we said, your node program will respond to **all** URL requests.

The `http.createServer` method gives you the ability to programmatically respond to URLs, but it's your responsibility to have your program do something useful for each and every request it receives.

Fortunately, much like PHP has frameworks that redirect all URLs to a main entry file (like `index.php`), javascript has its share of frameworks and libraries to help out with URL routing. The most popular of these is a package called Express. Before we can talk about Express, we'll need to talk about package management in Javascript.

Javascript Package Management

NodeJS ships with a package/dependency manager tool named `npm`. This is similar to systems like Ruby's Gems, Perl's CPAN, Java's maven, and most pertinent to us, PHP's Composer.

If you're familiar with Composer, you should *mostly* feel right at home with NPM. There are a few differences, the biggest of which is where NPM gets its packages from.

Both Composer and NPM have a default central repository/registry, with the option to create your own self-hosted repositories/registries. However, Composer's central repository (at packagist.org) contains *only* package meta-data, and only *points* to a package's ultimate home (an archive file, a GitHub repository, an svn repository, etc). The NPM registry (at registry.npmjs.org, browsable via www.npmjs.com) actually hosts the files for a package itself.

This centralization is part of a CommonJS standard on how registries should work. It's also an interesting case study on how a technical architecture choice can shape the sort of organization and business that grows around a piece of software — but that's another story for another time.

For our purposes, NPM is the de-facto central repository of javascript code for the entire world. We're going to use NPM to install a package named Express. Express is a simple web framework that sits on top of node's `http` library.

We'll get started by creating a file named `package.json` with the following contents

```
//File: package.json
{
    "dependencies":{
        "express": "^4.15.2"
```

```
    }
}
```

This file is similar to `composer.json` for PHP developers. The code above tells NPM our project is dependent on the `express` package. The version string (`^4.15.2`) is the version of the package NPM should download, and the `^` character is a special SemVer character that tells NPM what versions of Express it's OK to automatically upgrade to. Versioning and SemVer are a little beyond the scope of this article, but you can read more about SemVer on the NPM site.

With the above in place, run the `npm install` command

```
$ npm install
```

Your computer will stop and think for a bit while it calculates your dependencies and downloads all the needed Express package files, as well as any files Express itself is dependent on. Once that's complete, `npm` will write out a dependency tree describing the packages it downloaded.

```
$ npm install
/path/to/our-express-project
└─┬ express@4.15.2
  ├─┬ accepts@1.3.3
  │ ├─┬ mime-types@2.1.15
  │ │ └── mime-db@1.27.0
  │ └── negotiator@0.6.1
  ├── array-flatten@1.1.1
  ├── content-disposition@0.5.2
  ├── content-type@1.0.2
```

```
├── cookie@0.3.1
├── cookie-signature@1.0.6
├─┬ debug@2.6.1
│ └── ms@0.7.2
├── depd@1.1.0
├── encodeurl@1.0.1
├── escape-html@1.0.3
├── etag@1.8.0
├─┬ finalhandler@1.0.2
│ ├─┬ debug@2.6.4
│ │ └── ms@0.7.3
│ └── unpipe@1.0.0
├── fresh@0.5.0
├── merge-descriptors@1.0.1
├── methods@1.1.2
├─┬ on-finished@2.3.0
│ └── ee-first@1.1.1
├── parseurl@1.3.1
├── path-to-regexp@0.1.7
├─┬ proxy-addr@1.1.4
│ ├── forwarded@0.1.0
│ └── ipaddr.js@1.3.0
├── qs@6.4.0
├── range-parser@1.2.0
├─┬ send@0.15.1
│ ├── destroy@1.0.4
│ ├─┬ http-errors@1.6.1
│ │ └── inherits@2.0.3
│ └── mime@1.3.4
├── serve-static@1.12.1
```

```
├── setprototypeof@1.0.3
├── statuses@1.3.1
├─┬ type-is@1.6.15
│ └── media-typer@0.3.0
├── utils-merge@1.0.0
└── vary@1.1.1
```

You can find these installed packages in the `node_modules` folder.

```
$ ls -1 node_modules/
accepts
array-flatten
content-disposition
/* ... */
express
/* ... */
unpipe
utils-merge
vary
```

For PHP developers, this is similar to Composer's `vendor` folder. With Express in place, we're ready to create our first server side javascript application with routing.

Hello Express

We're going to start over with a new javascript program called `hello-express.js`. Create the following file in the same folder as your `package.json`

```
//File: hello-express.js
console.log("Started Program");
```

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

app.get('/', function (request, response) {
    console.log("Handling Request");
    response.send('Hello Express');
    console.log("Done Handling Request");
})

app.listen(8000);
console.log("Ended Program");
```

Once you have your program in a file, run it with the `node` command

```
$ node hello-express.js
```

Once your program is running, you should be able to load the following URL and see a page with the `Hello Express` text.

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

While this program looks very similar to our previous examples, there's a few important differences. First, we're requiring a different module

```
//File: hello-express.js
var express = require('express');
```

The module `express` is **not** a part of the NodeJS standard library. It's a module we installed when we ran `npm install`, and its source code lives at `node_modules/express/index.js`.

The next difference is this line

```
//File: hello-express.js
var app = express();
```

Unlike the `http` module from our first program, the `express` module returns *a function* instead of *an object*. You call this function to get an instance of the Express router object. Modules that return functions instead of objects may be a surprise to you, depending on which languages you've previously programmed in.

Finally, we have this block of code

```
//File: hello-express.js
app.get('/', function (request, response) {
    console.log("Handling Request");
    response.send('Hello Express');
    console.log("Done Handling Request");
})

app.listen(8000);
```

Here we setup a callback function, and start listening on port 8000. While similar to our `http` example, there's one important difference. The `get` method of the router object (in the `app` variable) accepts a *URL path* as its first argument. This callback method will *only* work for the `/` URL path. if you try to load a different URL

```
$ curl -i 'http://localhost:8000/foo'
HTTP/1.1 404 Not Found
X-Powered-By: Express
```

```
Content-Security-Policy: default-src 'self'
X-Content-Type-Options: nosniff
Content-Type: text/html; charset=utf-8
Content-Length: 142
Date: Thu, 11 May 2017 03:16:17 GMT
Connection: keep-alive

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Cannot GET /foo</pre>
</body>
</html>
```

you'll get a 404 response. Also, you'll notice we used the `get` method of the router object in the `app` variable. This means if we try to `POST` to our URL (as in `<form method="post">...</form>`, we'll also get a 404 error.

```
//a POST request, which is fine
$ curl -i 'http://localhost:8000/'
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 13
ETag: W/"d-oPRzYb9qK1AQJa1lUfQSoZqXcws"
Date: Thu, 11 May 2017 03:19:00 GMT
```

```
Connection: keep-alive

//a POST, which is not fine
$ curl -i -d foo:bar 'http://localhost:8000/'
HTTP/1.1 404 Not Found
X-Powered-By: Express
Content-Security-Policy: default-src 'self'
X-Content-Type-Options: nosniff
Content-Type: text/html; charset=utf-8
Content-Length: 140
Date: Thu, 11 May 2017 03:19:07 GMT
Connection: keep-alive

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Cannot POST /</pre>
</body>
</html>
```

For PHP developers, this may seem a bit weird. While some PHP frameworks allow you to setup URLs that only respond to a specific HTTP verb (`GET`, `POST`, etc.), a framework's main PHP file will always respond to a URL request, irrespective of the HTTP `Request Method` header.

In Express, if we want a URL that responds to `POST` requests, we'll need to use the `post` method of the router object.

```
//File: our-express-project/hello-express.js
app.get('/', function (request, response) {
    console.log("Handling Request");
    response.send('Hello Express');
    console.log("Done Handling Request");
});

//new post method
app.post('/', function (request, response) {
    console.log("Handling Request");
    response.send('This handles a post');
    console.log("Done Handling Request");
});
```

With the above method in place, we can now send a POST request to the `/` URL.

```
$ curl -i -d foo:bar 'http://localhost:8000/'
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 19
ETag: W/"13-RJ8nuMGjYVGWD9CkkYg15oj+HEw"
Date: Thu, 11 May 2017 03:23:26 GMT
Connection: keep-alive

This handles a post
```

HTML and More NPM

One thing you may have noticed with the above example is, we're not actually rendering HTML. This is another area where a PHP developer will need to make some adjustments in their expectations. Unlike most other programming languages, a PHP source file *starts* as an HTML template. It's only by invoking the `<?php` tag that the file enters PHP mode, and the parser starts looking for PHP code.

This leads many PHP programmers to eschew separate template systems and stick to PHP files. Also, even programmers who do use PHP template systems like twig, blade, or smarty will sometimes `include` in a quick and dirty PHP template when they need to.

As a javascript programmer, you don't have this option. You'll either need to build your HTML strings by hand, or download one of the many javascript template engines. We're going to download a template system named pug using npm.

To do this, we *could* edit our `package.xml` file manually to include the pug dependency along with a version number. However, this seems like work that's best left to a computer. The `npm` developers agree, and included command line flags that will let us automatically add a new package to our dependency list. Just type

```
$ npm install pug --save
```

The `npm` command will think for a bit, and then download the `pug` package (and all its dependencies) to the `node_modules` folder. Also, because we used the `--save` option, node automatically added `pug` to our `package.json` file

```
//File: our-express-project/package.json
{
  "dependencies": {
```

```
    "express": "^4.15.2",
    "pug": "^2.0.0-rc.1"
  }
}
```

This is nice for a few reasons. First, the `--save` option allow us to get out of the business of editing our `package.json` file directly. Second — `npm` *automatically* picks the latest version for us (`2.0.0-rc.1`) *and* a semantic versioning prefix (`^`).

If you're a little more conservative with your semantic versioning prefixes, you can tell `npm` to use something different

```
npm install pug --save --save-prefix='~'
```

We could also install a specific version of the package like this

```
npm install pug@0.1.0 --save
```

If you're not familiar with semantic versioning, when you use the `^` prefix

```
"some-module": "^2.0.0"
```

you're telling `npm` it's OK to update the package automatically if the package maintainer releases a version 2.1, 2.2, 2.3, etc. If you use the `~` prefix, you're telling `npm` it's OK to update the package if version `2.0.1`, `2.0.2`, `2.0.3`, etc. is available.

The semantic versioning system in NPM is a topic unto itself — for those already acquainted with the topic javascript's culture seems to be

> run the latest version of everything and let the chips fall where they may

While yolo may be how modern javascript rolls, realize that these version numbers do have consequences whenever you run `npm install`. For example if (in a separate folder)

```
$ cd ..
$ mkdir some-other-folder
```

you create the following `package.json` file

```
//File: some-other-folder/package.json
{
    "dependencies": {
        "express": "^4.14.0"
    }
}
```

and run `npm install` — you **will not** get version `4.14.0` of the package. You'll get the latest `4.*.*` version.

For our purposes, we'll stick with the version generated by `npm install pug --save`

Render a Template

Now that pug's installed, we can actually use it. First, lets create a simple pug template

```
//File: page.pug
<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
<h1>#{message}</h1>
</body>
</html>
```

Then, we'll modify our program to render this template, and then send the rendered HTML back.

```
//File: hello-express.js
console.log("Started Program");
var express = require('express');
var pug     = require('pug');

var app = express();


app.get('/', function (request, response) {
    console.log("Handling Request");
    var html = pug.renderFile('page.pug',{
        'message':'Hello Pug!'
    });
    response.send(html);

    console.log("Done Handling Request");
});
```

```
app.listen(8000);
console.log("Ended Program");
```

If you run the program and access our `http://localhost:8000` URL, you should see the rendered HTML. There's a few new things to cover here.

First, we've required in the `pug` module with this line

```
//File: hello-express.js
var pug     = require('pug');
```

Then, we use the pug module to load the `page.pug` template, and swap in the `message` variable. Unlike the `express` module (which returns a function), the `pug` module returns a traditional object with methods.

```
//File: hello-express.js
var html = pug.renderFile('page.pug',{
    'message':'Hello Pug!'
});
```

This is how you render a template in `pug`. Different template systems may offer different syntax for the same thing. The same is true for pug's template syntax

```
//File: page.pug
<h1>#{message}</h1>
```

We use the `#{variable-name}` syntax not because of anything to do with node or Express, but because this is how the pug package wants its variables.

Wrap Up

So that's the basics of routing and rendering in a server side javascript application. We have, of course, only scratched the surface. Experienced developers are probably yelling at ~~us~~ their screens about how Express supports views, which allows you to abstract your templates away, or how we ignored pug's advanced syntax for complex inheritance, or how we ignored simpler routing solutions, or etc.

Unlike the world of PHP frameworks, javascript projects tend to be more granular about picking and choosing different components to build (what amounts to) a custom framework on each project. Regardless of whether you see this as a good thing or a bad thing, it's how things are in javascript land.

Now that we have a bit of NodeJS and NPM under our belts, next time we're going to look at client side (i.e. browser) code in the age of modern javascript.

Series Navigation

<< Modern Javascript for PHP Developers

Client Side Javascript, Modules, and Webpack >>