Client Side Javascript, Modules, and Webpack

This entry is part 3 of 4 in the series Modern Javascript for PHP Developers. Earlier posts include Modern Javascript for PHP Developers, and Express and NPM for PHP Developers. Later posts include Build Watchers, and NPM as a Build Tool.

Earlier articles in this series included server side javascript that looked like this

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

That is, we used a function named `require` to load a *javascript module*. A module is a way to organize and load other javascript code in a way that doesn't conflict with the code in your current file.

As PHP programmers, we don't get code modules.

PHP's `include` and `require` statements allow us to organize our code into files, but those files are all mashed together in one global namespace. PHP namespaces came close to giving PHP programmers modules, but incomplete functionality and the general direction of the class oriented framework/autoloader culture means most PHP programmers use classes, not modules, to get code from different sources to play nice together.

NodeJS's implementation of the CommonJS module standard gives server side javascript developers a clear way to organize their code. Browser based developers, on the other hand, are stuck with a situation that's closer to PHP. Consider something like this

```
<script src="path/to/file.js"></script>
<script type="text/javascript">
    //... more code ...
</script>
<script src="path/to/file-2.js"></script>
```

Here, the browser loads each javascript file, and mashes them together into a single program along with the inline blocks. Client side (i.e. "in browser") javascript developers are left to their own devices as to how to protect their code from naming collisions, with only functional scope to save them

```
//a global variable
var someVariable = 'foo';

var main = function()
{
    var someVariable;   //a local function, only avaiable in this functic
                        //and any inner functions defined

    var main2 = function()
    {
        alert(someVariable); // from the main function
        alert(window.someVariable); // from the global scope
    }
```

```
    //referenced without first using `var` keyword?  you get the gloabl
    someOtherGlobal;
}
```

All is not lost though. If you're willing to give up the immediacy of working directly in your browser, modern javascript offers you a way to use CommonJS modules, as well as newer module systems, in your browser based programs.

Today we're going to explore one of those methods: a "transpiler" called webpack.

Hello Webpack

We're going to jump right in with a simple example. Just treat this like a copy/paste/cookbook section for now. Don't worry, we'll eventually explain everything we've done.

First, we're going to create a new node based project and use NPM to pull down the webpack package

```
$ mkdir webpack-tutorial
$ cd webpack-tutorial
$ echo '{}' > package.json
$ npm install webpack --save
$ ls -ld node_modules/webpack
```

Then, with webpack installed, we'll make a source folder for our client code

```
$ mkdir src-client
```

and a target folder for our web accessible assets.

```
$ mkdir pub
```

Next, we'll create an HTML file that pulls in a javascript file named `bundle.js` (via the `<script src="bundle.js"></script>` tag)

```
<!-- File: pub/example.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>Our Example File</title>

</head>
<body>
    <h1>Our Example File</h1>
    <script src="bundle.js"></script>
</body>
</html>
```

If you open this file in your browser, (either directly on disk or over the web via an HTTP URL), your browser will complain about a missing `bundle.js` file.

> Failed to load resource: `net::ERR_FILE_NOT_FOUND` bundle.js

This is expected.

With this scaffolding in place, we're going to create our program. However, we will not (at first) create a `bundle.js` file. Instead, we'll create a file that will serve as our program's main entry point. Create the following file in the following folder.

```
//#File: src-client/entry-point.js
helloWorld = require('./hello-world');

if(typeof alert !== 'undefined'){
    alert(helloWorld.getMessage());
}
else{
    console.log(helloWorld.getMessage());
}
```

You'll see this module imports a module named `hello-world` via `require`. Next, **we'll define** that `hello-world` module. Create the following file in the following folder.

```
//#File: src-client/hello-world.js
var toExport = {};
toExport.getMessage = function(){
    return 'Hello Webpack';
}
module.exports = toExport;
```

At this point, we're ready to run our program.

Running the Program

If you've been following along with this series, you know we can run this program from the command line like this

```
$ node src-client/entry-point.js
Hello Webpack
```

However, if we tried running `entry-point.js` in a browser, we'd get the following error

```
Uncaught ReferenceError: require is not defined
```

As we said earlier, there's no `require` function in native browser javascript. This is where webpack comes in. Run the following command from the root of your project folder

```
$ ./node_modules/webpack/bin/webpack.js src-client/entry-point.js pub/bur
Hash: e23306a81b5092dee254
Version: webpack 2.5.1
Time: 66ms
    Asset     Size  Chunks             Chunk Names
bundle.js  2.98 kB       0  [emitted]  main
   [0] ./src-client/hello-world.js 110 bytes {0} [built]
   [1] ./src-client/entry-point.js 162 bytes {0} [built]
```

You should now have a `bundle.js` file located at `pub/bundle.js`. If you open up (or reload) `example.html`, you should have a page that pops up an alert box with `Hello Webpack`.

Congratulations — you just *built* your first modern javascript client side program.

What Just Happened?

If we take a look at our program again

```
//#File: src-client/entry-point.js
helloWorld = require('./hello-world');

if(typeof alert !== 'undefined'){
    alert(helloWorld.getMessage());
}
else{
    console.log(helloWorld.getMessage());
}
```

We see the first line imports a module using a `./` path. This path indicates the module's source file is in the same folder as `entry-point.js` (vs. being in `node_modules`). The program then uses the `getMessage` method of the module to fetch a string for our `alert` call. If `alert` is not defined (i.e. we're running the program via the command line), we'll use the `console.log` method instead.

This is a pretty straight forward program. NodeJS knows to look for the `hello-world` module in a file named `hello-world.js`. If we look at this file

```
//#File: src-client/hello-world.js
var toExport = {};
toExport.getMessage = function(){
    return 'Hello Webpack';
}
module.exports = toExport;
```

we see a simple object definition, and a `getMessage` method added to that object. We also see something that may be new to you

```
module.exports = toExport;
```

The `module.exports` method is a NodeJS/CommonJS thing. This line basically says

> If a programmer imports `hello-world.js` as a module using the `require` function, that programmer should receive whatever is in `toExport` as a result

In other words, this is the line that tells javascript what object our module exports.

The end result of all this is a program we can run via `node`

```
$ node src-client/entry-point.js
Hello Webpack
```

However, as previously mentioned, we **cannot** run this program (as is) via a browser. To do that, we need something that will examine every file in our program, and create something that browser/client-side javascript understands. This is what webpack does.

Source to Source Compiler

Webpack is an example of a source to source compiler.

> A **source-to-source compiler**, **transcompiler** or **transpiler** is a type of compiler that takes the source code of a program written in one programming language as its input and produces the equivalent source code in another programming language.

In this case, our source language is a NodeJS program with CommonJS modules, and our target language is client-side compatible javascript. When we said

```
$ ./node_modules/webpack/bin/webpack.js src-client/entry-point.js pub/bur
```

we were running `webpack` on the command line.

Webpack's first argument is our source program

```
src-client/entry-point.js
```

Its second argument is the file that will contain the target program

```
pub/bundle.js
```

When we ran `webpack`, it parsed the code in our main entry point, looked for `require` statements, resolved those statements to files, and then parsed those files as well. If we had `required` in modules from our `node_modules` folder webpack would have scanned **those files** as well. Once everything is parsed, `webpack` created the browser compatible `bundle.js`.

If we look at the contents of this file

```
//#File: pub/bundle.js
/******/ (function(modules) { // webpackBootstrap
/******/    // The module cache
/******/    var installedModules = {};
/******/
```

```
/******/    // The require function
/******/    function __webpack_require__(moduleId) {
/******/
/******/        // Check if module is in cache
/******/        if(installedModules[moduleId]) {
/******/            return installedModules[moduleId].exports;
/******/        }
/******/        // Create a new module (and put it into the cache)
/******/        var module = installedModules[moduleId] = {
/******/            i: moduleId,
/******/            l: false,
/******/            exports: {}
/******/        };
/******/
/******/        // Execute the module function
/******/        modules[moduleId].call(module.exports, module, module.exp
/******/
/******/        // Flag the module as loaded
/******/        module.l = true;
/******/
/******/        // Return the exports of the module
/******/        return module.exports;
/******/    }
/******/
/******/
/******/    // expose the modules object (__webpack_modules__)
/******/    __webpack_require__.m = modules;
/******/
/******/    // expose the module cache
/******/    __webpack_require__.c = installedModules;
```

```
/******/
/******/    // identity function for calling harmony imports with the cor
/******/    __webpack_require__.i = function(value) { return value; };
/******/
/******/    // define getter function for harmony exports
/******/    __webpack_require__.d = function(exports, name, getter) {
/******/        if(!__webpack_require__.o(exports, name)) {
/******/            Object.defineProperty(exports, name, {
/******/                configurable: false,
/******/                enumerable: true,
/******/                get: getter
/******/            });
/******/        }
/******/    };
/******/
/******/    // getDefaultExport function for compatibility with non-harmc
/******/    __webpack_require__.n = function(module) {
/******/        var getter = module && module.__esModule ?
/******/            function getDefault() { return module['default']; } :
/******/            function getModuleExports() { return module; };
/******/        __webpack_require__.d(getter, 'a', getter);
/******/        return getter;
/******/    };
/******/
/******/    // Object.prototype.hasOwnProperty.call
/******/    __webpack_require__.o = function(object, property) { return (
/******/
/******/    // __webpack_public_path__
/******/    __webpack_require__.p = "";
/******/
```

```
/******/    // Load entry module and return exports
/******/    return __webpack_require__(__webpack_require__.s = 1);
/******/ })
/************************************************************************/
/******/ ([
/* 0 */
/***/ (function(module, exports) {

var toExport = {};
toExport.getMessage = function(){
    return 'Hello Webpack';
}
module.exports = toExport;



/***/ }),
/* 1 */
/***/ (function(module, exports, __webpack_require__) {

helloWorld = __webpack_require__(0);

if(typeof alert !== 'undefined'){
    alert(helloWorld.getMessage());
}
else{
    console.log(helloWorld.getMessage());
}
```

```
/***/ })
/******/ ]);
```

we see something that's not very human readable. This file isn't meant to be human readable or editable. Us humans edit files in `src-client` (or any folder we chose) using a javascript module system. Then, webpack creates something that a browser can understand.

Beyond CommonJS

Where `webpack` gets a little crazy is in its scope. Throughout the years there have been multiple attempts to create a javascript module standard. The standard used in NodeJS is called CommonJS.

Another standard is AMD. For folks coming from Magento 2, RequireJS is one of AMD's most well known implementations.

A third standard is the ES6/ES2015 import keyword, a part of the ongoing ECMAScript standards process.

The webpack project attempts to support **all** these standards. For example, we could create an entry point that uses an ES2015/ES6 module via the import statement.

```
//File: src-client/entry-point.js
import * as helloWorld from './hello-world2';
alert(helloWorld.getMessage());
```

and then define that module (using ES2015 syntax).

```
//#File: src-client/hello-world2.js
var getMessage = function(){
```

```
    return 'Hello ES2015';
};
export {getMessage};
```

This program **won't** run via NodeJS, because node doesn't (as of May 28, 2017) understand ES2015/ES6 modules.

```
$ node src-client/entry-point.js
/path/to/webpack/src-client/entry-point.js:2
import * as helloWorld from './hello-world2';
^^^^^^
SyntaxError: Unexpected token import
    at Object.exports.runInThisContext (vm.js:76:16)
    at Module._compile (module.js:528:28)
    at Object.Module._extensions..js (module.js:565:10)
    at Module.load (module.js:473:32)
    at tryModuleLoad (module.js:432:12)
    at Function.Module._load (module.js:424:3)
    at Module.runMain (module.js:590:10)
    at run (bootstrap_node.js:394:7)
    at startup (bootstrap_node.js:149:9)
    at bootstrap_node.js:509:3
```

However, if we build a `bundle.js` file via `webpack`

```
$ ./node_modules/webpack/bin/webpack.js src-client/entry-point.js pub/bur
```

we'll get a working program.

After this, things start getting weird. Consider the following program

```
//#File: src-client/entry-point.js

var helloWorld1 = require('./hello-world');
import * as helloWorld2 from './hello-world2';
alert(helloWorld1.getMessage());
alert(helloWorld2.getMessage());
```

This program uses **both** the CommonJS module loading syntax, as well as the ES2015 `import` module loading syntax. While this *might* be considered invalid syntax in some perfect future version of Javascript that implements all of ES2015, the `webpack` command can still handle it. The `webpack` transpiler will interpret the `require` call as a CommonJS module, and the `import` as a ES6 module.

If we had created a small RequireJS program in our main program

```
//#File: src-client/entry-point.js

var helloWorld1 = require('./hello-world');
import * as helloWorld2 from './hello-world2';
alert(helloWorld1.getMessage());
alert(helloWorld2.getMessage());

//a RequireJS program
require('./hello-world3', function(helloWorld){
    //...
});
```

The webpack transpiler would recognize this as RequireJS code and load `hello-world3.js`, looking for a `define` statements (per standard RequireJS).

While a powerful and pragmatic approach, this does create a weird situation where it's possible for your team to create javascript programs that **will only** run with webpack. Put another way, webpack lets you use modern, ES2015 modules, but your program ends up being something that's *not quite* ES2015.

Strict ES2015

There's one last thing to look out for here. When `webpack` sees the `import` keyword

```
import * as helloWorld2 from './hello-world2';
```

it automatically puts javascript into `"use strict"` mode. This can create some subtle problems. Consider the following program

```
//#File: src-client/entry-point.js

helloWorld1 = require('./hello-world');
import * as helloWorld2 from './hello-world2';
alert(helloWorld1.getMessage());
alert(helloWorld2.getMessage());

require('./hello-world3', function(helloWorld){
    //...
});
```

This is almost identical to our previous program, and `webpack` will compile this without issue. However — when we try to run this program in a browser, we'll get the following

error

```
bundle.js:99 Uncaught ReferenceError: helloWorld1 is not defined
    at Object.<anonymous> (bundle.js:99)
    at __webpack_require__ (bundle.js:20)
    at toExport (bundle.js:66)
    at bundle.js:69
```

Folks familiar with `"use strict"` will quickly spot the problem. The one change we made to our entry point was here

```
helloWorld1 = require('./hello-world');
```

We removed the `var` keyword. In non-strict javascript, that means `helloWorld1` is now a global variable. When modern javascript is running in strict mode, it will refuse to define a global variable. This might catch you off guard if you're used to the variable scoping in languages with formal modules (such as `ruby` or `python`).

Now that we have a better idea of what webpack does, lets take a look at how the modern front end javascript developer adds familiar libraries to their code.

NPM and Front End Libraries

Consider the following program

```
jQuery(function($){
    $('body').html('<h1>Hello World</h1>');
});
```

This program uses the jQuery library to replace the entire DOM `<body/>` contents with the string *Hello World*. If we were traditional javascript developers, we'd include jQuery with code that looked something like this

```
//#File: pub/hello-jquery.html

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>

    <script src="/path/to/jquery.js"></script>
    <script src="/path/to/our/hello-jquery.js"></script>
</body>
</html>
```

That is, we'd include the jQuery source via a `<script/>` tag. Then, we'd include our program's source via a `<script/>` tag. Or maybe we'd put this in the `<head/>`, or maybe we'd try to use RequireJS and use client side modules.

A modern javascript developer takes a different approach. They

1. Include the jQuery package in their `npm` project
2. Write their program to import jQuery using CommonJS, RequireJS, or `import` module syntax
3. Build a single javascript file for their HTML page

Lets give this a try. First, we'll add jQuery to our `npm` based project.

```
$ npm install jquery --save
/path/to/webpack-tutorial
└── jquery@3.2.1

npm WARN webpack No description
npm WARN webpack No repository field.
npm WARN webpack No license field.
```

If we peek at our `package.json` file, we'll see jQuery added to the list of dependencies

```
//#File: package.json
"dependencies": {
    "jquery": "^3.2.1",
    "webpack": "^2.5.1"
},
```

and we'll see there's a `jquery` folder in the `node_modules` folder

```
$ ls -1d node_modules/jquery/
node_modules/jquery/
```

Next, we'll write our program's main entry point.

```
//#File: src-client/hello-jquery.js
var jQuery = require('jQuery');
jQuery(function($){
```

```
    $('body').html('<h1>Hello jQuery</h1>');
});
```

Then, we'll use `webpack` to compile/transpile our program.

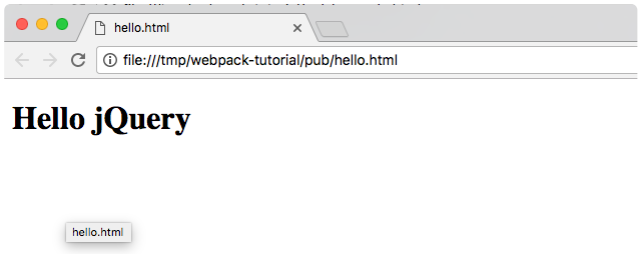
```
$ ./node_modules/webpack/bin/webpack.js src-client/hello-jquery.js pub/bu
Hash: 0b2ca6986e03d5a05a5c
Version: webpack 2.5.1
Time: 342ms
     Asset    Size  Chunks                    Chunk Names
bundle2.js  271 kB       0  [emitted]  [big]  main
   [0] ./~/jQuery/dist/jquery.js 268 kB {0} [built]
   [1] ./src-client/hello-jquery.js 99 bytes {0} [built]
```

Finally, we'll create a new HTML file that pulls in our just created `bundle2.js` file.

```
//#File: pub/hello.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>

<script src="bundle2.js"></script>
</body>
</html>
```

If you followed all the steps above, load up your `hello.html` file. You should see a simple HTML page with a *Hello jQuery* `<h1/>` title.



While this is a painfully simple example, many modern javascript frameworks follow a similar workflow, and some (like React) contain features that **require** you to use a transpiler.

Wrap Up

Phew! That was a lot — we touched on three different modern javascript module systems (CommonJS, AMD, ES2015 `import`), the concept of transcompilation, a specific transcompiler (webpack), and how to use all of them on client-side, browser based, projects.

Like before though, we've only scratched the surface. Webpack also offers the ability to automatically include modules written in javascript variants like coffee script, and features a CSS build system as well. If you're curious about its full capabilities, we recommend heading over to the official docs.

Also — it's important to note that that webpack is one of many transpiler/build tools used in the modern javascript world. While webpack is currently the bell of the ball, other projects may use older transpilers and there are sure to be new transpilers in the future. Not only has the traditional javascript developer lost their "edit a file, reload the page" workflows, they've inherited a complex series of build tools not maintained by any central authority.

Next time we're going to take a look at how NPM is the central pillar of this brave new build based world, as well as explore some ways that modern javascript tries to bring back the quick edit/refresh cycle us old timers are familiar with.

Series Navigation

<< Express and NPM for PHP Developers Build Watchers, and NPM as a Build Tool >>