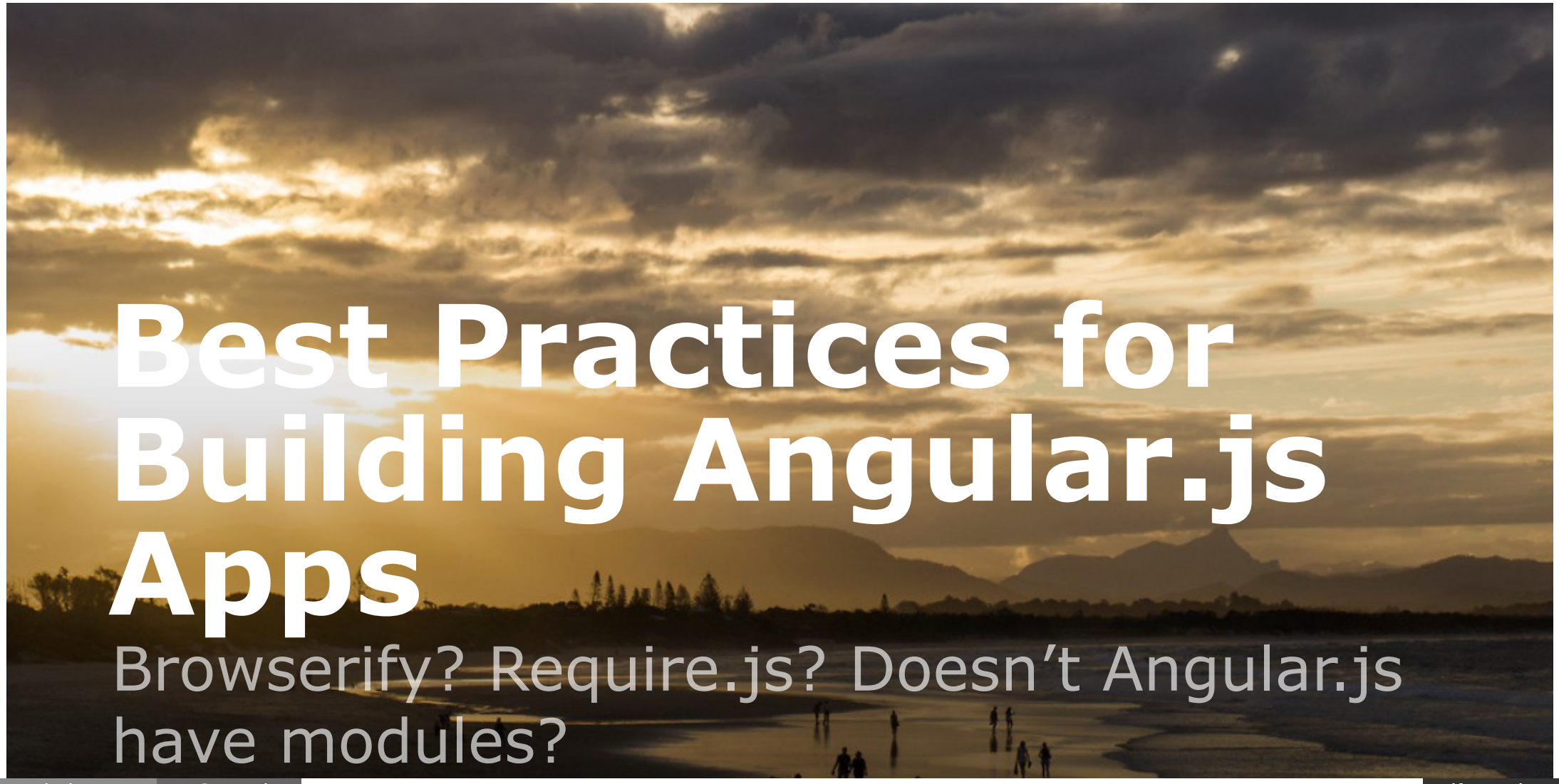
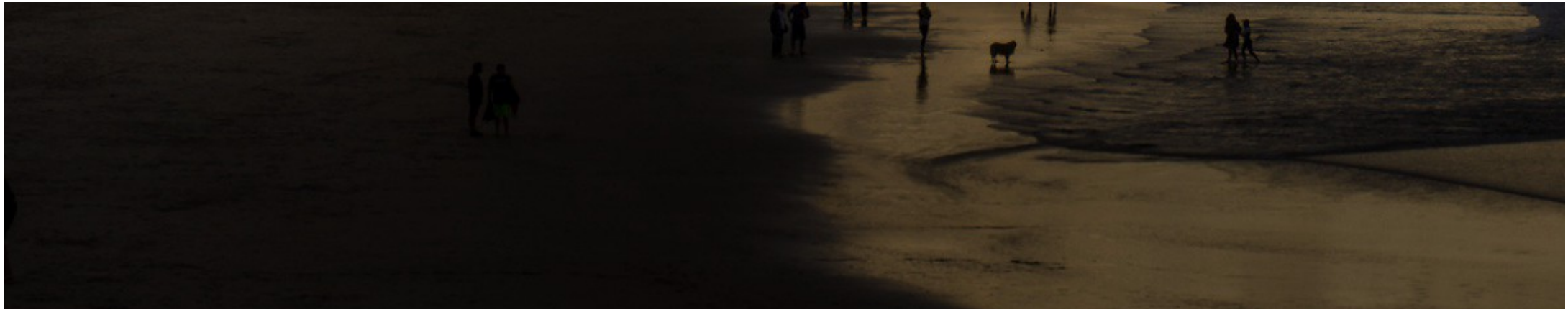
Jeff Dickey

Jul 9, 2014 · 13 min read



Best Practices for Building Angular.js Apps

Browserify? Require.js? Doesn’t Angular.js have modules?



Burke Holland had a fantastic post explaining how Angular loads an application and comparing the merits of browserify vs require.js in an Angular app.

I've worked with Angular on quite a few apps at this point, and have seen many different ways to structure them. I'm writing a book on architecting Angular apps right now with the MEAN stack and as such have researched heavily into this specific topic. I think I've set on a pretty specific structure I'm very happy with. It's a simpler approach than what Burke Holland has proposed.

I must note that if I was on a project with his structure, I would be content. It's good.

Before we start though, the concept of modules in the world of Angular can be a bit confusing, so let me lay out the current state of affairs.

What modules are in JavaScript

JavaScript comes with no ability to load modules. A "module" means different things to different people. For this article, let's use this definition:

> *Modules allow code to be compartmentalized to provide logical separation for the developers. In JavaScript, it also prevents the problem of conflicting globals.*

People new to JavaScript get a little confused about why we make such

a big deal about modules. I want to make one thing clear: **Modules are NOT for lazy-loading JavaScript components when needed**. Require.js *does* have this functionality, but that is not the reason it is important. Modules are important to due to the language not having support for it, and JavaScript desperately needing it.

A module can be different things. It could be Angular, lodash (you're not still using underscore, are you?), shared code in your organization, some gist you found online, or separating features out inside your codebase.

JavaScript doesn't support modules, so we've traditionally had a few various approaches. (Feel free to skip this next section if you understand JavaScript modules)

.noConflict()

Let me illustrate the problem. Let's say you want to include jQuery in your project. jQuery will define the global variable '$'. If, in your code, you have an existing variable '$' those variables will conflict. For years, we got around this problem with a .noConflict() function. Basically .noConflict() allows you to change the variable name of the library you're using.

If you had this problem, you would use it like this:

```
<script>
var $ = 'myobject that jquery will conflict with'
</script>
<script src='//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js'></script>
<script>
// $ is jQuery since we added that script tag
var jq = jQuery.noConflict();
// $ is back to 'myobject that jquery will conflict with'
// jq is now jQuery
</script>
```

noConflict.html hosted with by **GitHub**

This has been a common practice in most JavaScript libraries, but it's not a fantastic solution. It doesn't provide very good compartmentalizing of code, it forces you to declare things before you use them, and it requires the imported code (either a library or your own code) to actually implement a .noConflict() function.

If that's confusing, read up on it. It's important to understand the problem before you continue onto the solutions below.

Nobody was happy with .noConflict(), so they started looking into other ways to solve the problem. We have 4 solutions worth mentioning in this context:

- Require.js (Implementation of AMD)
- Browserify (Implementation of CommonJS)

- Angular dependency injection
- ES6 modules

Each one has its pros and cons, and each works quite a bit differently. You can even use 1 or 2 in tandem (Burke used 2). I'll cover what each does, how they work with Angular, and which one I suggest.

Sample App

Let's get a little Angular app together so we can talk about it.

Here is a simple app that lists users off Github.
The code is here, but it's the completed version we will build in this post. Read through for no spoilers!

All the JavaScript could be in this one file:

```
1  var app = angular.module('app', [])
```

```
2
3  app.factory('GithubSvc', function ($http) {
4  return {
5  fetchStories: function () {
6  return $http.get('https://api.github.com/users')
7      }
8    }
9  })
10
11 app.controller('GithubCtrl', function ($scope, GithubSvc) {
12   GithubSvc.fetchStories().success(function (users) {
```

Initial Angular app with all code in one file

First we declare an ‘app’ object that is our module. We then define a service ‘GithubSvc’ with one function that can serve us users from Github.

After that, we define a controller that uses the service to load that array into $scope. (This is the HTML page that renders it)

Splitting into separate files

The trouble is that this code is all in one file. Totally unreasonable for a real app. Maybe I'm a curmudgeon, but when I first started looking at Angular and the code samples all showed how to do this, all I wanted to see was a real world solution with proper separation.

I would like to have this code in a structure like this:

```
src/module.js
src/github/github.svc.js
src/github/github.ctrl.js
```

Note: If this app got large, it might make sense to have a separate 'github' module as well.

The alternate way to do this would be to split things out by functionality rather than part of the codebase:

```
src/module.js
src/services/github.svc.js
src/controllers/github.ctrl.js
```

I don't have a strong preference either way. Probably *very* large apps would benefit from the former, and smaller ones the latter.

Regardless, without using a module loader like browserify or require.js, we would have to add a script tag for every one of these files. That's a no go. That could easily grow to hundreds of files.

There are performance reasons why you don't want to have tons of script tags too. The browser does pipeline them, but it can only do so many at a time. They have overhead, and the latency would be killer to our friends outside of California.

So here is the goal:

We need a way to have many Angular files in dev, but they need to be loaded into the browser in bulk (not a script tag for each one).

This is why people look to module loaders like require.js or browserify. Angular allows you to logically separate out code, but not files. I'm going to show an easier way, but first let's examine the available module loaders.

Require.js□—□Too complicated

Require.js was the first major push towards coming up with a consistent way to have modules inside of JavaScript. Require.js allows you to define dependencies inside a JavaScript file that you depend on. It runs inside the browser and is capable of loading modules as needed.

It accomplishes 2 general tasks, loading of modules and handling the load order.

Unfortunately it's really complicated to setup, requires your code to be written in a specific way, certainly has the steepest learning curve, and can't deal with circular dependencies well□—□and that can happen when trying to use a module system on top of Angular.

Burke Holland covered the issues with using require.js with Angular very well, so I encourage you to read that for a clearer reason why you should not use Angular with require.js.

> *Working with RequireJS and AngularJS was a vacation on Shutter Island. On the surface everything looks very normal. Under that surface is Ben Kingsley and a series of horrific flashbacks.*□—□*Burke Holland*

The ability for require.js to load modules on demand is also something that won't work with Angular (at least, in a reasonable situation). That seems to be something people want, but I've certainly never worked on a project that needed it.

I want to emphasize that last point as people get this wrong: Module systems are **not** so that you only load the code you need. Yes require.js does do that, but it's not *why* require.js is useful. Modules are useful to logically separate code for developers to reason about it easier.

In any case, it's a bad solution and I won't show you how to do it. I bring it up because people often ask me how to integrate require.js with Angular.

Browserify □—□ A much better module loader

Where require.js has the browser load the modules, browserify runs on the server before it runs in the browser. You can't take a browserify file and run it in a browser, you have to 'bundle' it first.

It uses a similar format (and is almost 100% compatible with) the Node.js module loading. It looks like this:

```
1  var moduleA = require('my-module')
2  var moduleB = require('your-module')
3
4  moduleA.doSomething(moduleB)
```

commonjs.js hosted with by **GitHub**

Browserify example

It's a really pretty, easy to read format. You simply declare a variable and 'require()' your module into it. Writing code that exports a module is very easy too.

In Node, it's great. The reason it can't work in the browser, however, is that it's synchronous. The browser would have to wait when hitting one of those require sections, then make an http call to load the code in. Synchronous http in a browser is an absolute no-no.

It works in Node since the files are on the local filesystem, so the time it takes to do one of those 'requires()' is very fast.

So with browserify, you can take code like this and run it with browserify and it will combine all the files together in a bundle that the browser can use. Once again, Burke's article covers using browserify with Angular very well.

By the way, if everything I just said about browserify is confusing, don't worry about it. It's certainly more confusing than the solution I'm about to propose.

It is a great tool I would jump to use on a non-Angular project. With Angular, however, we can do something simpler.

Angular Dependency Injection □ — □ Solves most of our problems

Go back and look at our sample app's app.js. I want to point out a couple of things:

It doesn't matter what order we create the service or the controller. Angular handles that for us with its built-in Dependency Injection. It also allows us to do things like mocking out the service in a unit test. It's great, and my number one favorite feature inside Angular.

Having said that, with this method, we **do** need to declare the module first to use that 'app' object. It's the only place that order of declarations matter in Angular, but it's important.

What I want to do, is simply concatenate all the files together into one, then require just that JavaScript file in our HTML. Because the app object has to be declared first, we just need to make sure that it's declared before anything else.

Gulp Concat

To do this, I will be using Gulp. Don't worry about learning a newfangled tool though, I'm going to use it in a very simple way and you can easily port this over to Grunt, Make, or whatever build tool you want (shockingly, even asset pipeline). You just need something that can concat files.

I've played around with all the popular build systems and Gulp is far and away my favorite. When it comes to building css and javascript, specifically, it's bliss.

You might be thinking I'm just replacing one build tool (browserify) with another (gulp), and you would be correct. Gulp, however, is much more general purpose. You can compose this Gulp config with other tools like minification, CoffeeScript precompilation (if you're into that sort of thing), sourcemaps, rev hash appending, etc. Yes it's nothing browserify can't do, but once you learn how to do it with Gulp you can do the same on any other asset (like css). Ultimately it's much less to learn.

You can use it to process png's, compile your sass, start a dev node server, or running any code you can write in node. It's easy to learn, and will provide a consistent interface to your other developers. It provides us a platform to extend on later.

I would much rather just type ‘gulp watch’ and have that properly watch all my static assets in dev mode than have to run ‘watchify’, a separate node server, a separate sass watcher, and whatever else you need to keep your static files up to date.

First I’ll install Gulp and gulp-concat (gotta be in the project *and* global):

```
$ npm install --global gulp
$ npm install --save-dev gulp gulp-concat
```

By the way, you’ll need a package.json in your app and have Node installed. Here’s a little trick I do to start my Node apps (npm init is too whiny):

```
$ echo '{}' > package.json
```

Then toss in this gulpfile.js:

```
var gulp = require('gulp')
var concat = require('gulp-concat')

gulp.task('js', function () {
  gulp.src(['src/**/module.js', 'src/**/*.js'])
    .pipe(concat('app.js'))
    .pipe(gulp.dest('.'))
})
```

gulpfile.js hosted with by **GitHub**

gulpfile.js

This is a simple task that takes in the JavaScript files in src/ and concatenates them into app.js. Because it expects this array, any file named module.js will be included first. Don't worry too much about understanding this code, when we get to minification I'll clear it up.

If you want to play along at home, use these files, then run 'gulp js' to build the assets. Donezo.

For more on Gulp, read my article on setting up a full project with it

Icky Globals

We can do better. You know how you create that 'app' variable? That's a global. Probably not a problem to have one 'app' global, but it might be a problem when we grow to have more and more modules, they may conflict.

Luckily Angular can solve this for us very easily. The function angular.module() is both a getter and a setter. If you call it with 2 arguments:

```
1  angular.module('app', ['ngRoute'])
```

module.js hosted with by **GitHub**

angular.module as a setter

That's a setter. You just created a module 'app' that has 'ngRoute' as a dependency. (I won't be using ngRoute here, but I wanted to show what it looks like with a dependent module)

Calling that setter will also return the module as an object (that's what we put into var app). Unfortunately you can only call it once. Disappointingly, getting this stuff wrong throws nasty error messages that can be frustrating to newbies. Stick to the xxx method and all will be good though.

If we call angular.module() with a single argument:

```
1  angular.module('app')
```

module.js hosted with by **GitHub**

angular.module getter

It's a getter and also returns the module as an object, but we can call it as many times as we want. For this reason, we can rewrite our components from this:

```
1  app.factory('GithubSvc', function ($http) {
2  return {
```

```
3   fetchStories: function () {
4   return $http.get('https://api.github.com/users')
5       }
6     }
7   })
```

github.svc.js hosted with by **GitHub**

Global module service

Into this:

```
1   angular.module('app')
2   .factory('GithubSvc', function ($http) {
3   return {
4   fetchStories: function () {
5   return $http.get('https://api.github.com/users')
6       }
7     }
8   })
```

github.svc.js hosted with by **GitHub**

No globals involved

The difference is *subtle* and might seem innocuous to new JavaScript developers. The advanced ones are nodding along now though. **To maintain a large JavaScript codebase is to prevent the usage of globals.**

To you pedants: I realize that there is still a global ‘angular’ object, but there’s almost certainly no point in avoiding that.

Here we have a pretty well functioning way to build the assets, but there are a few more steps we need to get to the point of a fine-tuned build environment. Namely, it’s a pain to have to run ‘gulp js’ every time we want to rebuild ‘app.js’.

Gulp Watch

This is really easy, and I think the code speaks for itself (Lines 10-12):

```
1  var gulp = require('gulp')
2  var concat = require('gulp-concat')
3
4  gulp.task('js', function () {
5    gulp.src(['src/**/module.js', 'src/**/*.js'])
6      .pipe(concat('app.js'))
7      .pipe(gulp.dest('.'))
8  })
9
10 gulp.task('watch', ['js'], function () {
11   gulp.watch('src/**/*.js', ['js'])
12 })
```

Gulp with watching

This just defines a ‘gulp watch’ task we can call that will fire off the ‘js’ task every time a file matching ‘src/**/*.js’ changes. Blammo.

Minification

Alright, let’s talk minification. In Gulp we create streams from files

(gulp.src), then pipe them through various tools (minification, concatenation, etc), and finally output them to a gulp.dest pipe. If you know unix pipes, this is the same philosophy.

In other words, we just need to add minification as a pipe. First, install gulp-uglify to minify:

```
$ npm install -D gulp-uglify
```

```
var gulp = require('gulp')
var concat = require('gulp-concat')
var uglify = require('gulp-uglify')

gulp.task('js', function () {
  gulp.src(['src/**/module.js', 'src/**/*.js'])
    .pipe(concat('app.js'))
    .pipe(uglify())
    .pipe(gulp.dest('.'))
})
```

gulpfile.js hosted with by **GitHub**

But we have a problem! It has munged the function argument names Angular needs to do dependency injection! Now our app doesn't work. If you're not familiar with this problem, read up.

We can either use the ugly array syntax in your code, or we can introduce ng-gulp-annotate.

NPM install:

```
$ npm install -D gulp-ng-annotate
```

And here's the new gulpfile:

```
1 var gulp = require('gulp')
2 var concat = require('gulp-concat')
3 var uglify = require('gulp-uglify')
  var ngAnnotate = require('gulp-ng-annotate')
```

```
5
6  gulp.task('js', function () {
7    gulp.src(['src/**/module.js', 'src/**/*.js'])
8      .pipe(concat('app.js'))
9      .pipe(ngAnnotate())
10     .pipe(uglify())
11     .pipe(gulp.dest('.'))
12 })
```

Gulp minification with ng-annotate

I hope you're starting to see the value in Gulp here. How I can use a conventional format of Gulp plugins to quickly solve each of these build problems I am running into.

Sourcemaps

Everyone loves their debugger. The issue with what we've built so far is that it's now this minified hunk of JavaScript. If you want to console.log in chrome, or run a debugger, it won't be able to show you relevant info.

Here's a Gulp task that will do just that! (Install gulp-sourcemaps)

```
var gulp = require('gulp')
var concat = require('gulp-concat')
var sourcemaps = require('gulp-sourcemaps')
var uglify = require('gulp-uglify')
var ngAnnotate = require('gulp-ng-annotate')

gulp.task('js', function () {
  gulp.src(['src/**/module.js', 'src/**/*.js'])
    .pipe(sourcemaps.init())
      .pipe(concat('app.js'))
      .pipe(ngAnnotate())
      .pipe(uglify())
```

Sourcemaps!

Why Concat is Better

Concat works better here because it's simpler. Angular is handling all of

the code loading for us, we just need to assist it with the files. So long as we get that module setter before the getters, we have nothing to worry about.

It's also great because any new files we just add into the directory. No manifest like we would need in browserify. No dependencies like we would need in require.js.

It's also just generally one less moving part, one less thing to learn.

What we built

Here is the final code. It's an awesome starting point to build out your Angular app.

- It's got structure.
- It's got a dev server.

- It's got minification.
- It's got source maps.
- It's got style. (The Vincent Chase kind, not the CSS kind)
- It **doesn't** have globals.
- It **doesn't** have shitloads of <script> tags.
- It **doesn't** have a complex build setup.

I tried to make this not about Gulp, but as you can tell: I freaking love the thing. As I mentioned earlier, you could achieve a similar setup with anything that can concat.

If there is interest, I could easily extend this to add testing/css/templates/etc. I already have the code. EDIT: https://github.com/dickeyxxx/angular-boilerplate

Third-party code

For third-party code: if it's something available on a CDN (Google CDN, cdnjs, jsdelivr, etc), use that. If the user has already loaded it from another site, the browser will reuse it. They also have very long cache times.

If it's something not available on a CDN, I would still probably use a new script tag but load it off the same server as the app code. Bower is good for keeping these sorts of things in check.

If you have a lot of third-party code, you should look into minifying and concatenating them like above, but I would keep it separate from your app code so you don't have just one huge file.

ES6 Modules□—□The real solution

The next version of JavaScript will solve this problem with built-in modules. They worked hard to ensure that it works well for both fans of CommonJS (browserify) and AMD (require.js). This version is a ways out, and you probably won't be able to depend on the functionality without a shim of some kind for at least a year, probably a few. When it does come out, however, this post will be a relic explaining things you won't need to worry about (or at least it'll be horrifically incorrect).

Angular 2.0

It's worth mentioning that Angular 2.0 will use ES6 modules, and at that point we'll be in bliss. It's nowhere close to release though, so for now, if you want to use Angular, you need a different option. Angular 2.0 will be a dream. It's going to look a lot more like a series of useful packages than a framework, allowing you to pick and choose functionality, or bake them into an existing framework (like Ember or Backbone).

Angular 2.0 will use a separate library di.js that will handle all of this. It's way simpler, and it's only a light layer on top of ES6 modules. We should be able to easily use it in all apps, not just Angular apps. The unfortunate thing for you is that you will need to deal with the crufty state of affairs with JavaScript modules until then.

Man. I love all these great ways JavaScript is improving, but god damn is it a lot to keep learning.

If you'd like to learn more about Angular, check out my book on creating apps with the MEAN stack.

P.S. I have some code samples you can use to asynchronously load an Angular app. Any interest in reading about that? EDIT: https://github.com/dickeyxxx/ng-async

Jeff Dickey
CLI Engineer at Heroku

Follow