Building Angular 2 TypeScript Applications with Gulp

By Jeffry Houser

A DotComIt Whitepaper

Contents

I spent some time trying to build an Angular 2 TypeScript application with Gulp and Browserify. Unfortunately, the only way I could get it to work was to combine all the Angular 2 libraries into a single file with my custom code. Bundling Angular separately from my custom code would not load the finished application in the browser. I went back to the drawing board and found a new approach. This article will explain how.

You can find all the code behind this article in our GitHub account.

The Application

Before we start expanding on the build process, let's create a super simple, Hello World, application that uses Angular 2 and TypeScript.

The TypeScript files

Create an empty project directory, and then create a src folder inside it. The src directory will contain all the source code. When building Angular 2 applications, I use a directory structure that mimics the package structure I used to use when building older applications. Classes are put in a directory structure like 'com/dotComIt/*appName/somethingDescriptive*'. The appName will be a descriptive name for the application; I named it sample. The *somethingDescriptive* relates back to the purpose of the files in the folder. In some applications; I have named things after their type such as services, controllers, or views. In others I have named things after their purpose, such as login or task.

For this sample, I put the main TypeScript file in directory com/DotComIt/sample/main. The first file is main.ts:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
platformBrowserDynamic().bootstrapModule(AppModule);
```

This file does three things. First, it imports the Angular library, platform-browser-dynamic, and gives it the name of platformBrowserDynamic. Then it imports a custom library, app.module. We'll create that file next. Then it uses the two libraries to bootstrap the application, which is Angular magic for making the app work.

Next, create app.module.ts:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

This file imports two Angular libraries, the core library and the platform-browser library. It imports one custom library, app.component. Then a module is created. It imports the browserModule, declares the AppComponent, and bootstraps the AppComponent. The AppModule is exported, which is the code that allows the main.ts to access the AppModule.

Now, look at the app.component file:

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`,
})
export class AppComponent  { name = 'World'; }
```

This file imports the Angular core library. It creates a component; which defines the actual app. The selector refers to the HTML tag. The template defines an in-line template that will replace the tag after the application is bootstrapped.

That completes the TypeScript portion of the application

Configure SystemJS

The Angular TypeScript approach of using imports is not available as a native JavaScript construct. To get around this, a module loading system is needed. Angular 2 uses SystemJS. For this to work, SystemJS must be configured so that it knows where to find the Angular libraries, and our custom application. Create a file named systemjs.config.js in the src/js/systemJSConfig directory. When building applications; I like to keep my custom files separate from my library files; and the systemJS configuration should need to be set up once and will not need to change.

Setup the file with an Immediately Invoked Function Express (IIFE):

```
(function (global) {
})(this);
```

This is the JavaScript method to create a self-executing function. Inside the function, we want to configure SystemJS:

```
System.config({
});
```

The config() method is used to configure the SystemJS library. It takes an object, which is currently empty. Add in a paths configuration:

```
paths: {
  'js:': 'js/'
},
```

This configuration object tells the library that the js path will point to the js directory. When we write build scripts, we'll put all the relevant Angular libraries in the JS sub-directory of our final build.

Next, create a map configuration:

```
map: {
 app: 'com',
 '@angular/core': 'js:@angular/core/bundles/core.umd.js',
 '@angular/common': 'js:@angular/common/bundles/common.umd.js',
 '@angular/compiler': 'js:@angular/compiler/bundles/compiler.umd.js',
 '@angular/platform-browser':
                      'js:@angular/platform-browser/bundles/platform-browser.umd.js',

 '@angular/platform-browser-dynamic':
       'js:@angular/platform-browser-dynamic/bundles/platform-browser-dynamic.umd.js',

 '@angular/http': 'js:@angular/http/bundles/http.umd.js',
 '@angular/router': 'js:@angular/router/bundles/router.umd.js',
 '@angular/forms': 'js:@angular/forms/bundles/forms.umd.js',
 'rxjs': 'js:rxjs',
 'angular-in-memory-web-api':
                      'js:angular-in-memory-web-api/bundles/in-memory-web-api.umd.js'
},
```

The map configuration tells the code 'when we reference 'x' in code; go look for this library. The main angular libraries are listed. Explanation of the Angular libraries is beyond the scope of this article. The important things to notice are that the js path is specified in the file location for most of the libraries. The second thing to notice is that the app map points to our com directory, where all our custom code is located.

Next, specify the packages. A package is a collection of files that have shared functionality:

```
packages: {
  app: {
    main: './dotComIt/sample/main/main.js',
    defaultExtension: 'js'
  },
  rxjs: {
    defaultExtension: 'js'
  }
}
```

Two packages are specified. The first is app, which is our main application and custom code. It defines the main entry point, main.js—the name of the file that main.ts will be converted too. It also specifies the default extension of js. The second package specified is a package used by Angular, rxjs.

This is a lot of setup, but we're almost done and we can start focusing on the build scripts.

The Main Index Page

Let's look at the last bit of our application, the main index.html file. Create this file in the root of the src directory. It will be the page that users use to load our application. Start with a basic HTML page:

```
<html>
  <head>
    <title>Angular QuickStart</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
  </head>
  <body>
  </body>
</html>
```

This is a simple HTML page. Let's add some JavaScript script tags:

```
<script src="js/core-js/client/shim.min.js"></script>
<script src="js/zone.js/dist/zone.js"></script>
<script src="js/reflect-metadata/Reflect.js"></script>
<script src="js/systemjs/dist/system.src.js"></script>
```

The combination of these JavaScript libraries, and the other libraries specified in the SystemJS Config are what make Angular 2 applications work. A deeper explanation is beyond the scope of this article.

Next, load the systemjs.config.js file:

```
<script src="js/systemJSConfig/systemjs.config.js"></script>
```

Next, initialize the SystemJS App:

```
<script>
    System.import('app').then(null, console.error.bind(console));
</script>
```

This looks into the config file; finds the app, which points to our custom 'com' directory code; and then starts the code running.

The body needs the main application's tag:

```
<my-app>Loading AppComponent content here ...</my-app>
```

That completes the HTML portion of the application.

What's Missing?

We wrote a bunch of code, but we are far from a runnable application. First, the TypeScript must be compiled into JavaScript. Second, neither the Angular libraries, nor the JavaScript dependencies are anywhere yet. We have script tags, but no files to actually be loaded. The rest of this article will address those omissions.

The Setup

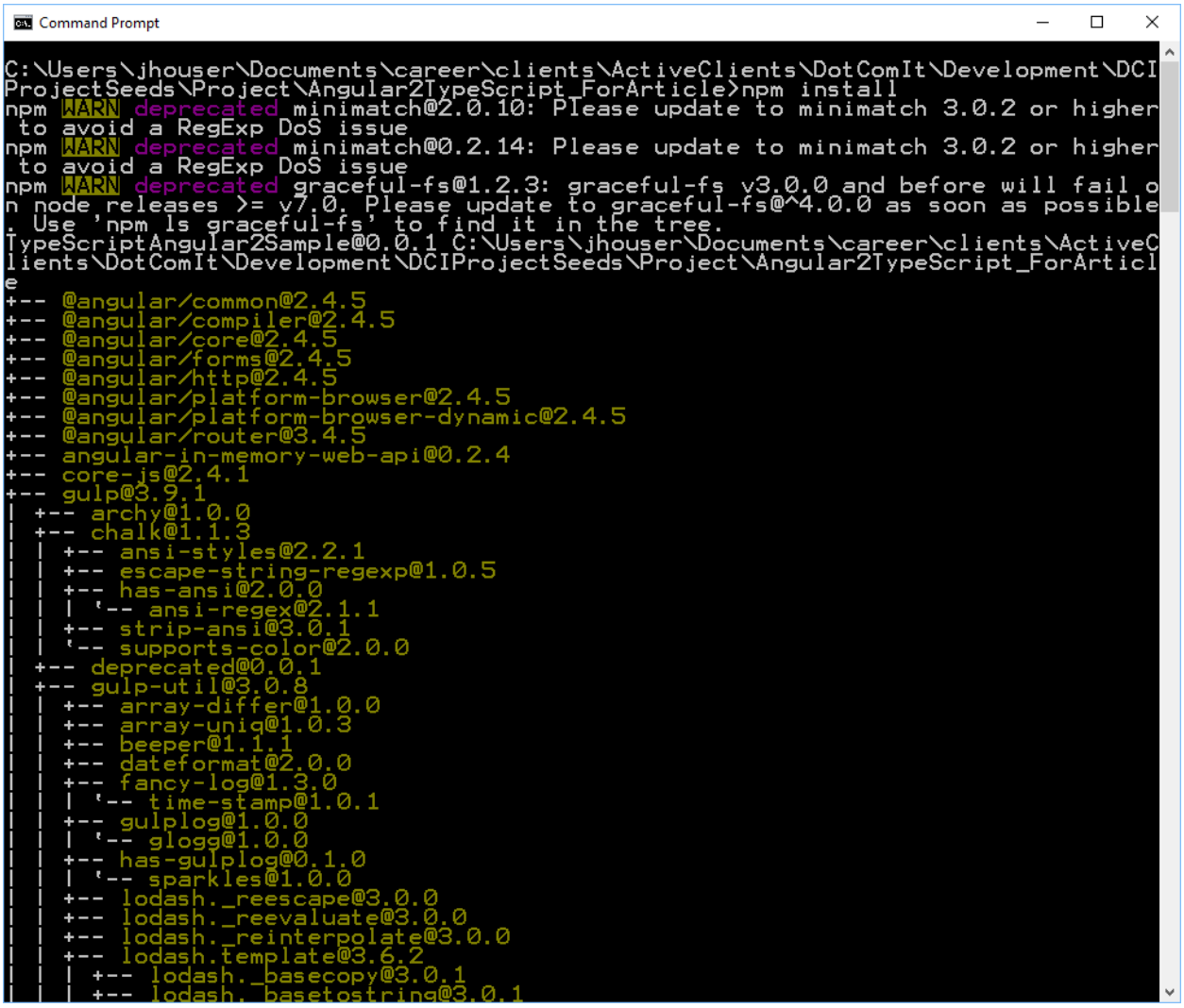
Gulp runs on top of NodeJS. If you don't already have it installed, do it now. The formal instructions or getting setup will be more helpful than anything I'd tell you in this article. After that, we need to create a package.json in the root directory. Copy and paste this one:

```
{
  "name": "TypeScriptAngular2Sample",
  "version": "0.0.1",
  "description": "TypeScript Angular 2 Sample for Article.",
  "author": "Jeffry Houser",
  "license": "ISC",
  "dependencies": {
    "@angular/common": "~2.4.0",
    "@angular/compiler": "~2.4.0",
    "@angular/core": "~2.4.0",
    "@angular/forms": "~2.4.0",
    "@angular/http": "~2.4.0",
    "@angular/platform-browser": "~2.4.0",
    "@angular/platform-browser-dynamic": "~2.4.0",
    "@angular/router": "~3.4.0",
    "angular-in-memory-web-api": "~0.2.2",
    "systemjs": "0.19.40",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.8",
    "rxjs": "5.0.1",
    "zone.js": "^0.7.4"
  },
  "devDependencies": {
    "gulp": "^3.9.1",
    "typescript": "^2.1.5"
  },
  "repository": {}
}
```

This includes all the major Angular 2 libraries and dependencies in the dependency section. The devDependencies section includes Gulp and TypeScript. Install all this stuff with one command:

```
npm install
```

You'll see something like this:



This creates a directory named node_modules that includes all the modules that NodeJS installed for us, along with any related dependencies. This is a base for creating the build scripts. As we need gulp plugins we'll install them individually.

Compiling TypeScript

This section will show you how to compile the TypeScript to JavaScript. First, it will be run through a Lint process that validates it for syntactical errors. Then we'll perform the actual compilation and then review the finished code.

Install Dependencies

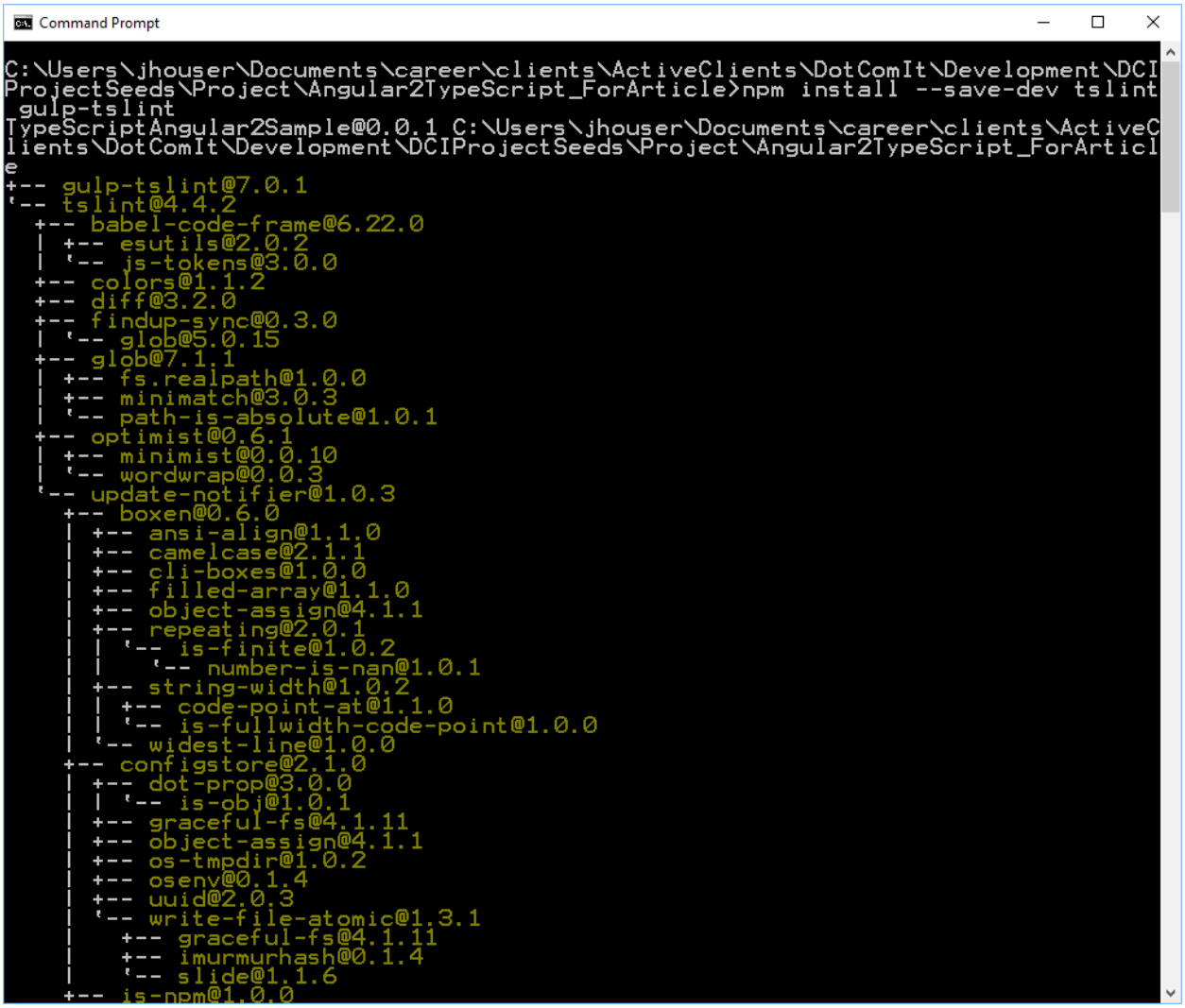
We'll need to install a few new NodeJS Modules for this section:

- **tslint**: A TypeScript linter
- **gulp-tslint**: A gulp plugin for tslint.
- **gulp-typescript**: A TypeScript plugin for Gulp

First install tslint and the gulp-tslint plugin:

```
npm install --save-dev tslint gulp-tslint
```

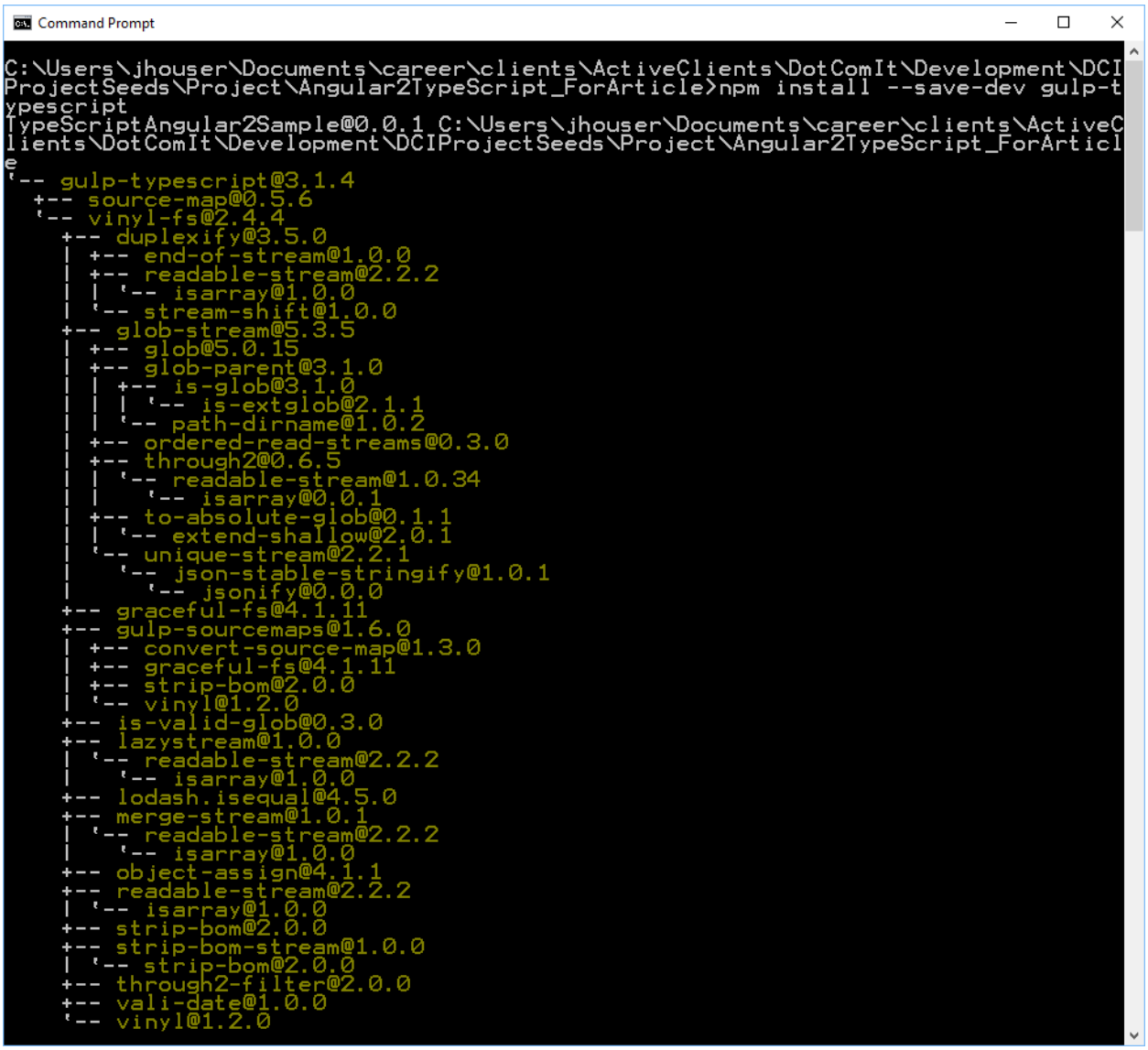
You should see results something like this:



Then install gulp-typescript:

```
npm install --save-dev gulp-typescript
```

You'll see results similar to this:



That constitutes dependencies, now it is time to setup the gulp file.

Linting the TypeScript

Create a file named gulpfile.js in the root directory of the project. Our first task to create is going to lint the TypeScript code. The first step is to import the gulp and gulp-tslint libraries:

```
var gulp = require("gulp");
var tslint = require('gulp-tslint');
```

This will make them available for use within our gulp script.

Next, I'm going to define the location of the typeScriptSource:

```
var sourceRoot = "src/";
var typeScriptSource = [sourceRoot + "**/*.ts"];
```

This is split up into two variables. The first just points to the source directory and that will be used throughout our script. The second uses the sourceRoot to create a wildcard glob that will cover all existing type script files in the main directory.

Now, create the gulp task to lint the TypeScript:

```
gulp.task('tslint', function() {
});
```

This is an empty task that does nothing. Use gulp.src() to tell gulp which items to process:

```
return gulp.src(typeScriptSource)
```

Then, run the tslint() task:

```
.pipe(tslint({
    formatter: 'prose'
}))
```

This will parse the TypeScript, and make a collection of any errors. Then, report the errors:

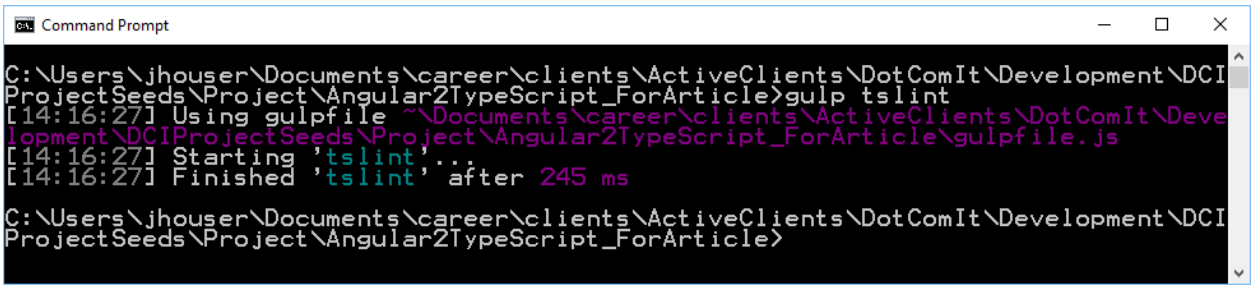
```
.pipe(tslint.report());
```

That is the completed task. Before we run it, we'll need to configure it. Specific configuration options are beyond the scope of this article. Put this tslint.json file, in your root directory and you'll be fine. The file comes from the official Angular 2 Quickstart documentation.

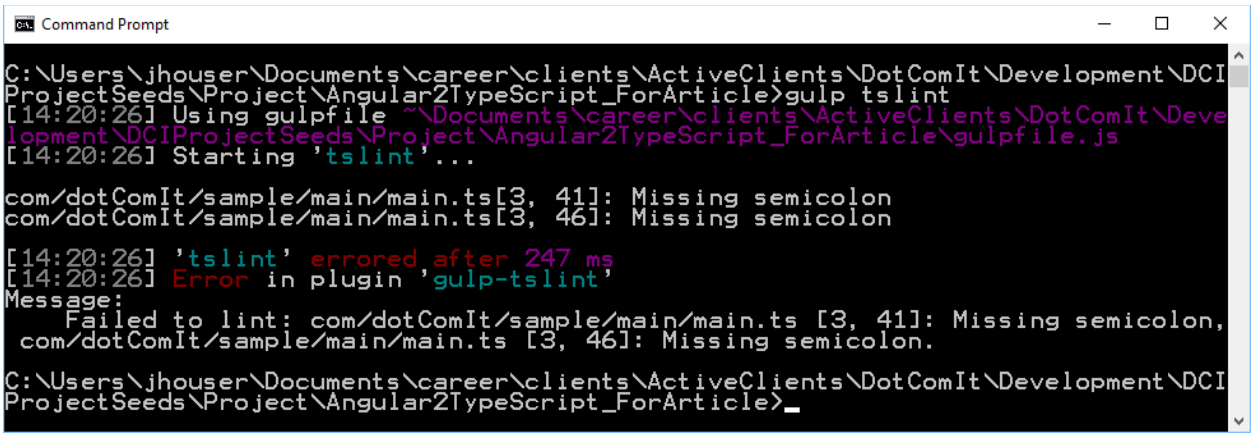
Run the task:

```
gulp tslint
```

You'll see something like this:



No issues. What happens when there are issues? I removed a semi-colon and added some random characters to the main.ts file and reran the lint process:



It correctly found the error. The lint process performs a syntax validation; it does not validate the code for accuracy of imports or other bugs.

Compile TypeScript to JavaScript

The next step is to turn the TypeScript into JavaScript using the gulp-typescript plugin. First, let's create some variables. First, the destination path for the output files:

```
var destinationPath = 'build';
```

I always put the processed files in a directory called build. Then, create an instance of the gulp-typescript module:

```
var tsc = require("gulp-typescript");
```

Use the gulp-typescript library to create a project:

```
var tsProject = tsc.createProject("tsconfig.json");
```

This refers to an external tsconfig.json file. I don't want to expand on the full details of the config file; but there are two important things I want to draw attention to. The first is a few items in the compilerOptions:

```
"compilerOptions": {
  "target": "es5",
  "module": "commonjs",
}
```

The module type is 'commonjs' which is the module type used behind the scenes by Angular 2. The second important thing here is the target, es5. This stands for EcmaScript 5; which JavaScript is an implementation of and most browsers support.

The second important item in the config file is:

```
"exclude": [
  "node_modules"
]
```

The exclude option tells the type script project not to process files in the node_modules directory. In essence, it ignores all the Angular libraries installed via NodeJS. These libraries already come with compiled bundles. We'll deal with moving those later.

Now, create a buildTS task:

```
gulp.task("buildTS", ["tslint"], function() {
});
```

This gulp task is named BuildTS. Before this task is run; the tslint task will execute. The main task first specifies the source:

```
return gulp.src(typeScriptSource)
```

It uses the same typeScriptSource variable that the tslint task used. Then, it adds the tsProject as a gulp pipe:

```
.pipe(tsProject())
```

Finally, it specifies the destination path:

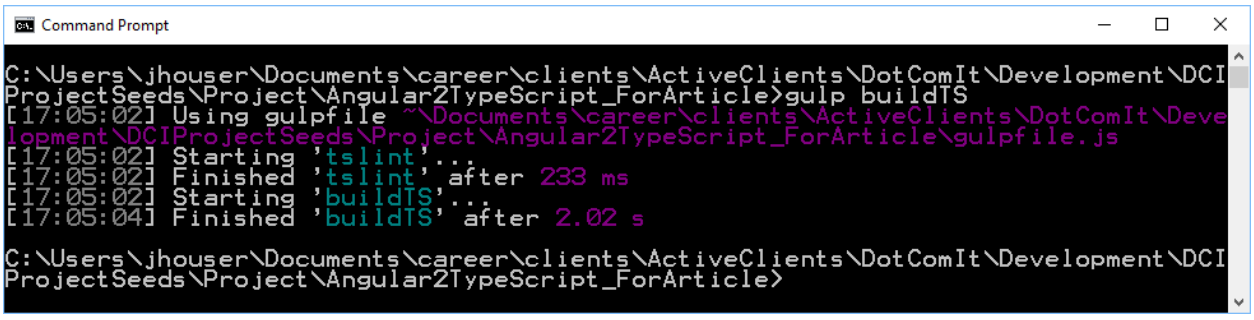
```
.pipe(gulp.dest(destinationPath));
```

What the full script does is take all the TypeScript files in our source directory, run them through the gulp-typescript compiler, and save the output to a build directory.

Run the script:

```
gulp buildTS
```

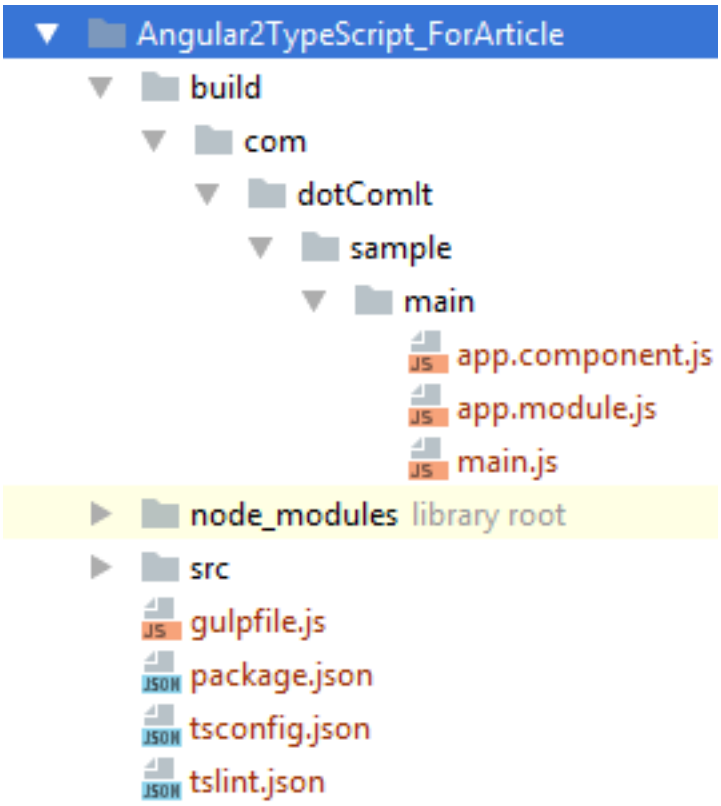
You'll see results similar to this:



You can see from the output that both the tslint and buildTS process ran. No errors; so let's review the output.

Review the JavaScript code

Look in your project directory and you'll see a brand new build directory:



Each TypeScript file was turned into a JavaScript file. Let's examine the main.js file; as it is simple:

```
"use strict";
var platform_browser_dynamic_1 = require("@angular/platform-browser-
dynamic");
var app_module_1 = require("./app.module");
platform_browser_dynamic_1.platformBrowserDynamic().bootstrapModule(app_modul
e_1.AppModule);
```

The first thing you'll notice is that the JavaScript iteration replaces the import statements with require statements. You'll recognize the require statement from the NodeJS code we are writing. However, browsers don't inherently support that. This is here because of commonJS module creation was specified in the compiler options. The systemJS library allows require() to be used in the browser, and that is what Angular 2 uses.

Beyond that, the JavaScript code is not all that different from the original TypeScript code. You can review the other two JS files and you'll find similar usage.

Copy JS Libraries and HTML, and other stuff

We aren't a spot where we have a runnable application yet, we still need to copy the HTML files, and the SystemJS configuration file from the src directory to the build directory. We also need to copy the Angular libraries from the node_modules directory to the build directory. Gulp makes it pretty easy to copy files.

Copy the Angular Libraries

There are a lot of required Angular libraries and we don't want to write a script for each one. Thankfully, we don't have to, as we can specify an array of glob directories to cover all the relevant libraries:

```
var angularLibraries = [
    'core-js/client/shim.min.js',
    'zone.js/dist/**',
    'reflect-metadata/Reflect.js',
    'systemjs/dist/system.src.js',
    '@angular/**/bundles/**',
    'rxjs/**/*.js',
    'angular-in-memory-web-api/bundles/in-memory-web-api.umd.js'
]
```

This covers all the JavaScript libraries required for Angular 2 applications such as zonejs and the shim library. It includes the Angular bundles, and the SystemJS library. Glob wildcards match files make it easy to find the relevant files needed.

Next, we'll need a destination path to put the libraries:

```
var destinationPathForJSLibraries = destinationPath + '/js';
```

I put all the external libraries in the 'js' directory as an organizational tactic. Now the actual task:

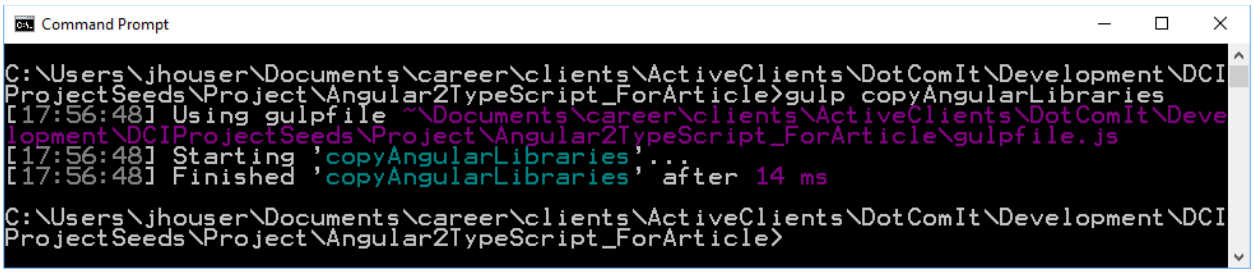
```
gulp.task('copyAngularLibraries', function () {
    gulp.src(angularLibraries, {cwd: "node_modules/**"})
        .pipe(gulp.dest(destinationPathForJSLibraries));
});
```

The task is named copyAngularLibraries. It is simple. It takes the array as a source. The cwd flag with the src specifies the current working directory, meaning all the libraries are located in node_modules. Then the task specifies the destination path. This will copy all the required angular files from the node_modules directory to the build/js directory.

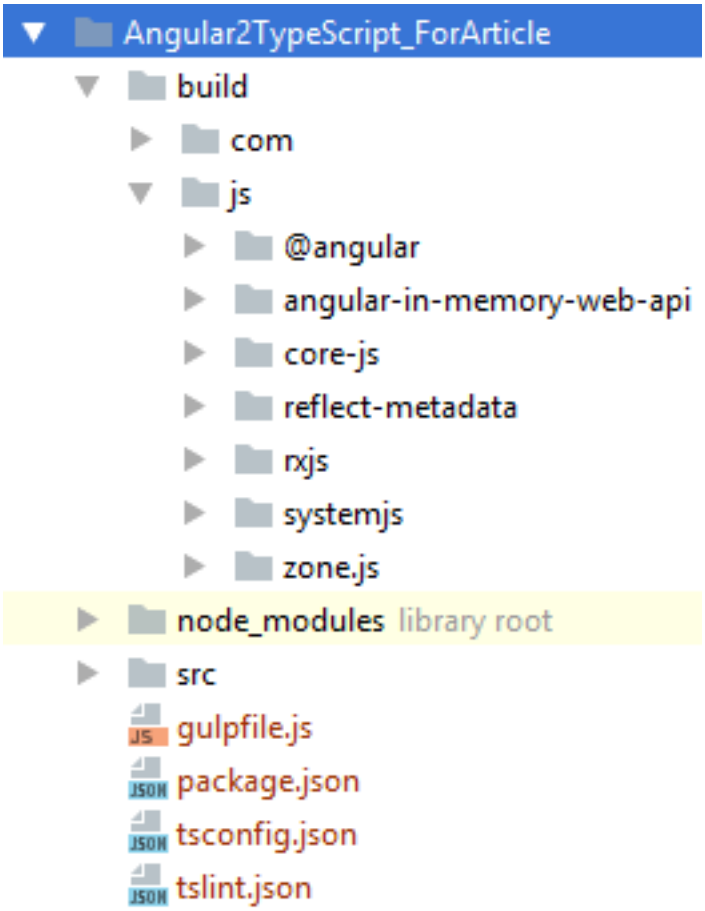
Run this task:

```
gulp copyAngularLibraries
```

You should see results like this:



You should see an updated build directory:



This is the first step, but we still need to copy other information.

Copy JS Libraries

If we have any JavaScript libraries put in the src/js folder we also want to copy those to the build directory. The SystemJS Configuration file is one example. The same approach can be used as with the Angular libraries, but is even simpler. First, create a glob array to find the JS files:

```
var javaScriptLibraries = [sourceRoot + 'js/**/*.js'];
```

Then create the task:

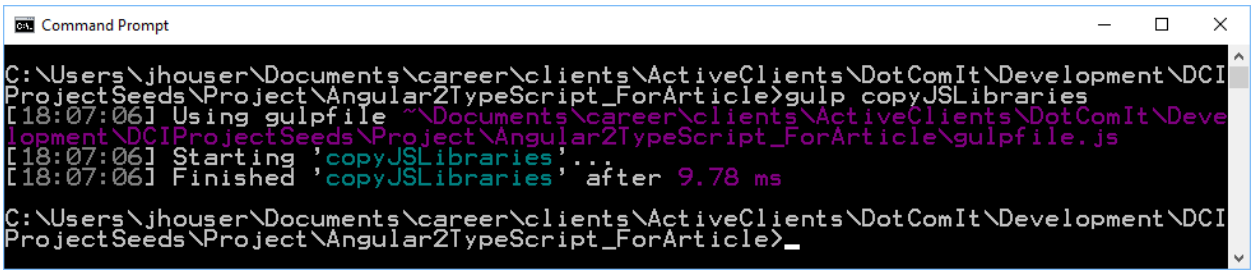
```
gulp.task('copyJSLibraries', function () {
    gulp.src(javaScriptLibraries)
        .pipe(gulp.dest(destinationPathForJSLibraries));
});
```

This is a simple task that specifies the input source and the destination. It uses the same destination variable that was used for the Angular libraries in the previous section.

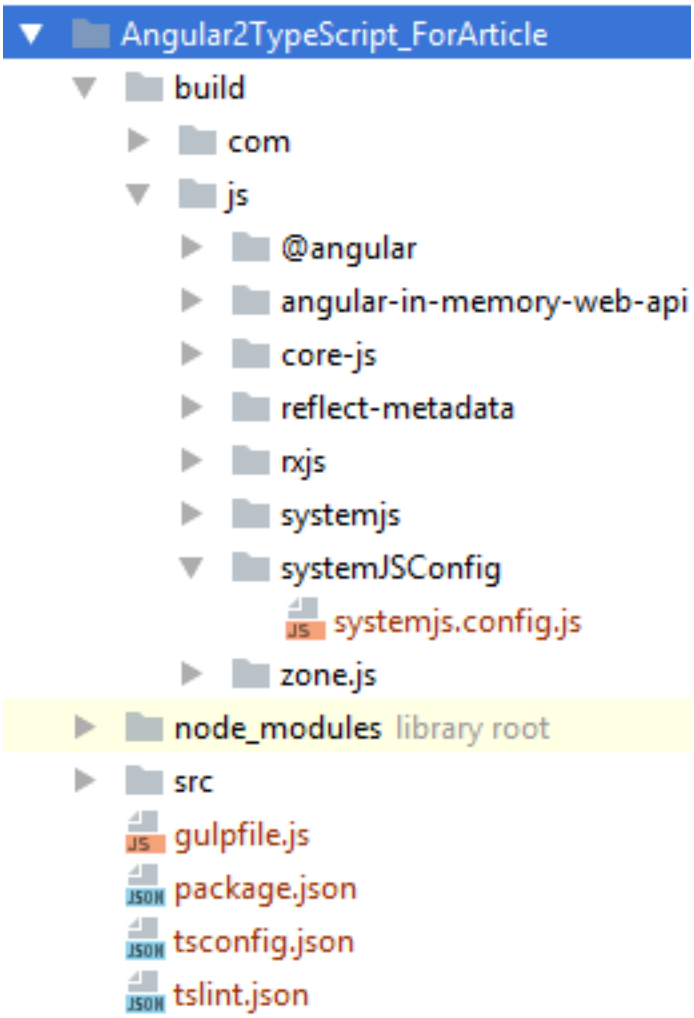
Run the task:

```
gulp copyJSLibraries
```

You'll see this:



Check the build directory and see the final file:



The systemjs.config.js file was successfully copied from the src directory to the build directory.

Copy HTML Files

The app currently only has a single HTML file, but it is likely that a full app will have more, especially if we start using external HTML templates. I want to write a task copy all HTML Files from the src directory to the build directory. First, create a glob array to find the HTML files:

```
var htmlSource = [sourceRoot + '**/*.html'];0
```

Then create the task:

```
gulp.task('copyHTML', function () {
    return gulp.src(htmlSource)
        .pipe(gulp.dest(destinationPath));
});
```

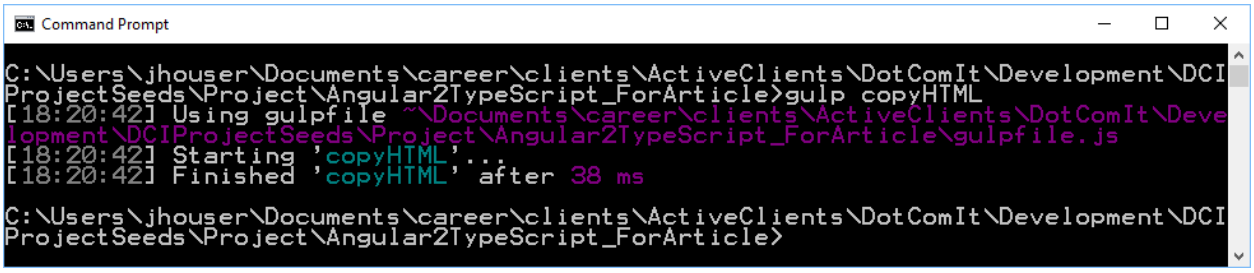
This is a simple task that specifies the input source and the destination. It uses the same destination variable that was used for the Angular libraries in the previous two sections. It could be easily combined

with the copyJSLibraries task, however I prefer to keep them separate for logistical reasons, since HTML files and JS files are different.

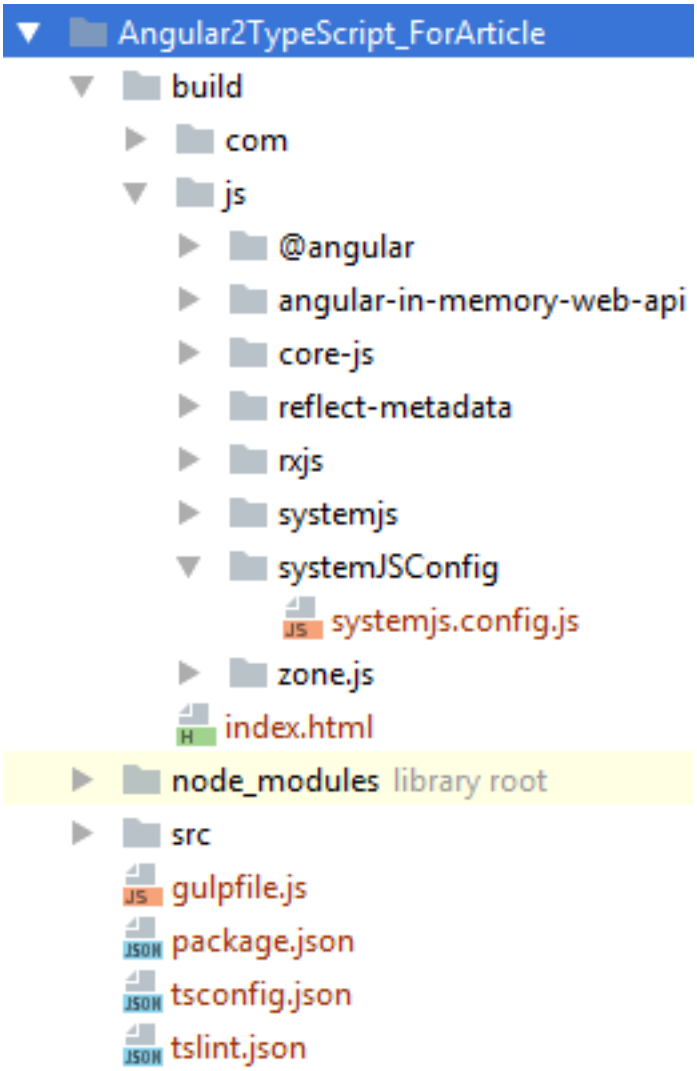
Run the task:

```
gulp copyHTML
```

You'll see this:

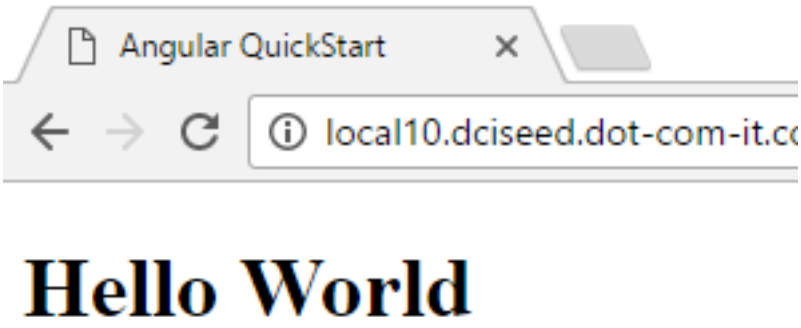


Check the build directory and see the final file:



The index.html file was successfully copied from the root of the src folder to the root of the build directory.

A side effect of copying the files is that you can run the app now:



Not an interesting app, but proof our scripts work.

Source Maps

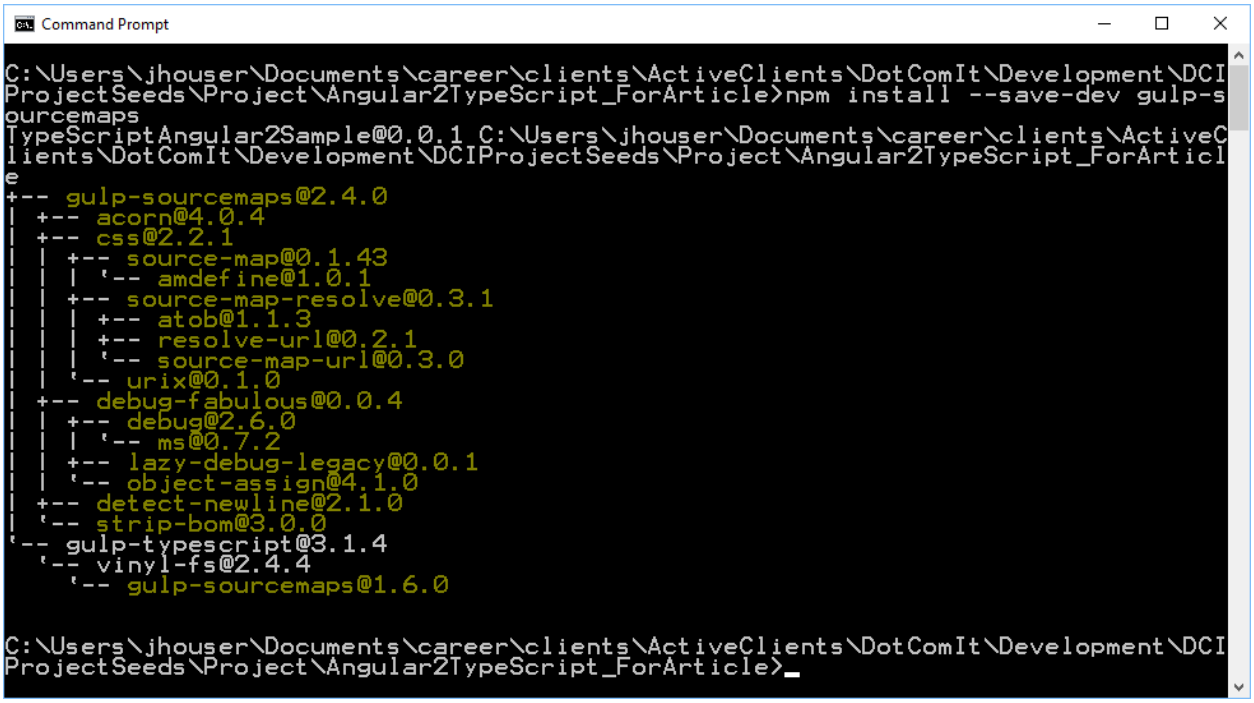
Since the TypeScript goes through a compilation process, the code running in the browser is not the same as the code we wrote. It has been translated from one language, TypeScript, to another, JavaScript. When an error occurs how can we debug it? The answer is to create a source map. We can look at the source maps in the browser, to learn where we went wrong in the TypeScript code. The usual browser debugging tools, such as break points and watches, will work with source maps.

Install Source Map Plugin

Gulp has a source map plugin that allows us to easily create source maps. First install it:

```
npm install --save-dev gulp-sourcemaps
```

You'll see something like this:



The plugin is now available to use within your gulp script.

Generate Source Maps

The first step is to load the source map plugin in your gulpfile.js:

```
var sourcemaps = require('gulp-sourcemaps');
```

I am going to add a configuration variable for the location of the maps:

```
var mapPath = 'maps';
```

I like to keep my maps separate from the generated code, but you do not have to specify a separate path if you do not prefer.

Now, you can add the source map generation to the buildTS task. First, here is the task unedited:

```
gulp.task("buildTS", ["tslint"], function() {
    return gulp.src(typeScriptSource)
        .pipe(tsProject())
        .pipe(gulp.dest(destinationPath));
});
```

There are two steps to the source map generation. The first is to initialize the maps. We want to do this as close to the front of the chain as possible so that the map code can keep track of all the changes. Then we want to save the maps. We want to do this as close to the end of the chain as possible. This is the modified task body:

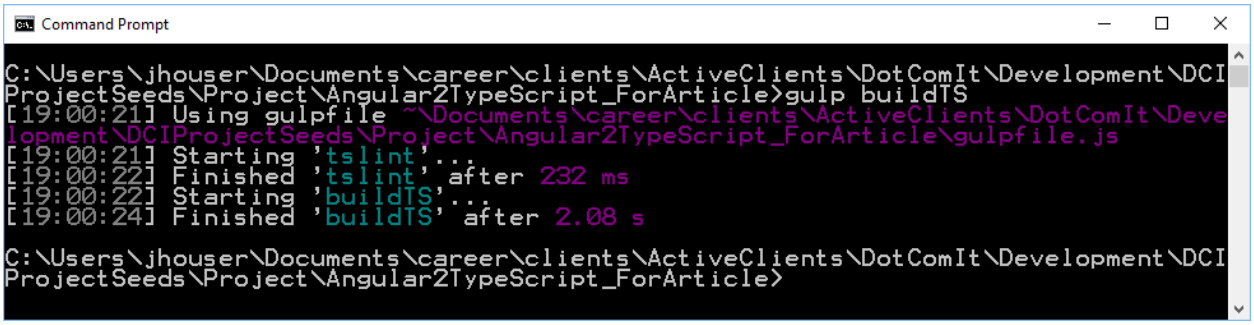
```
return gulp.src(typeScriptSource)
    .pipe(sourcemaps.init())
    .pipe(tsProject())
    .pipe(sourcemaps.write(mapPath))
    .pipe(gulp.dest(destinationPath));
```

The second pipe calls the init() method on the source map module. The second to last pipe saves the source maps with the write() method.

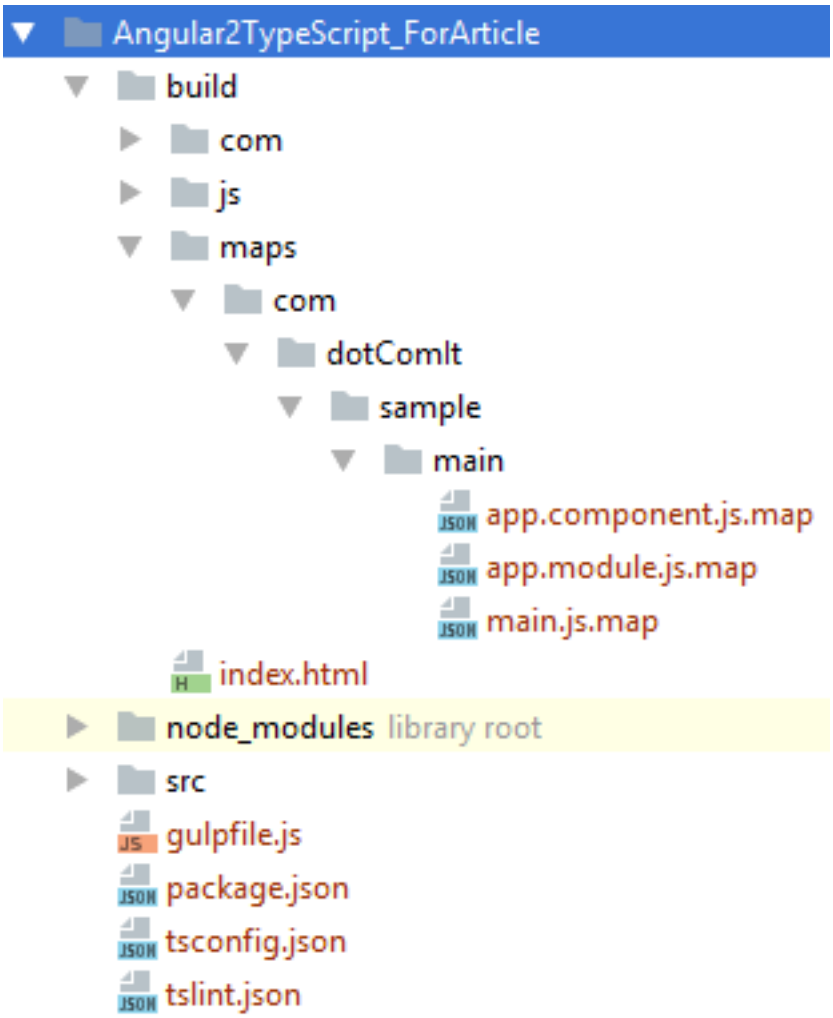
Run the task:

```
gulp buildTS
```

You'll see this:

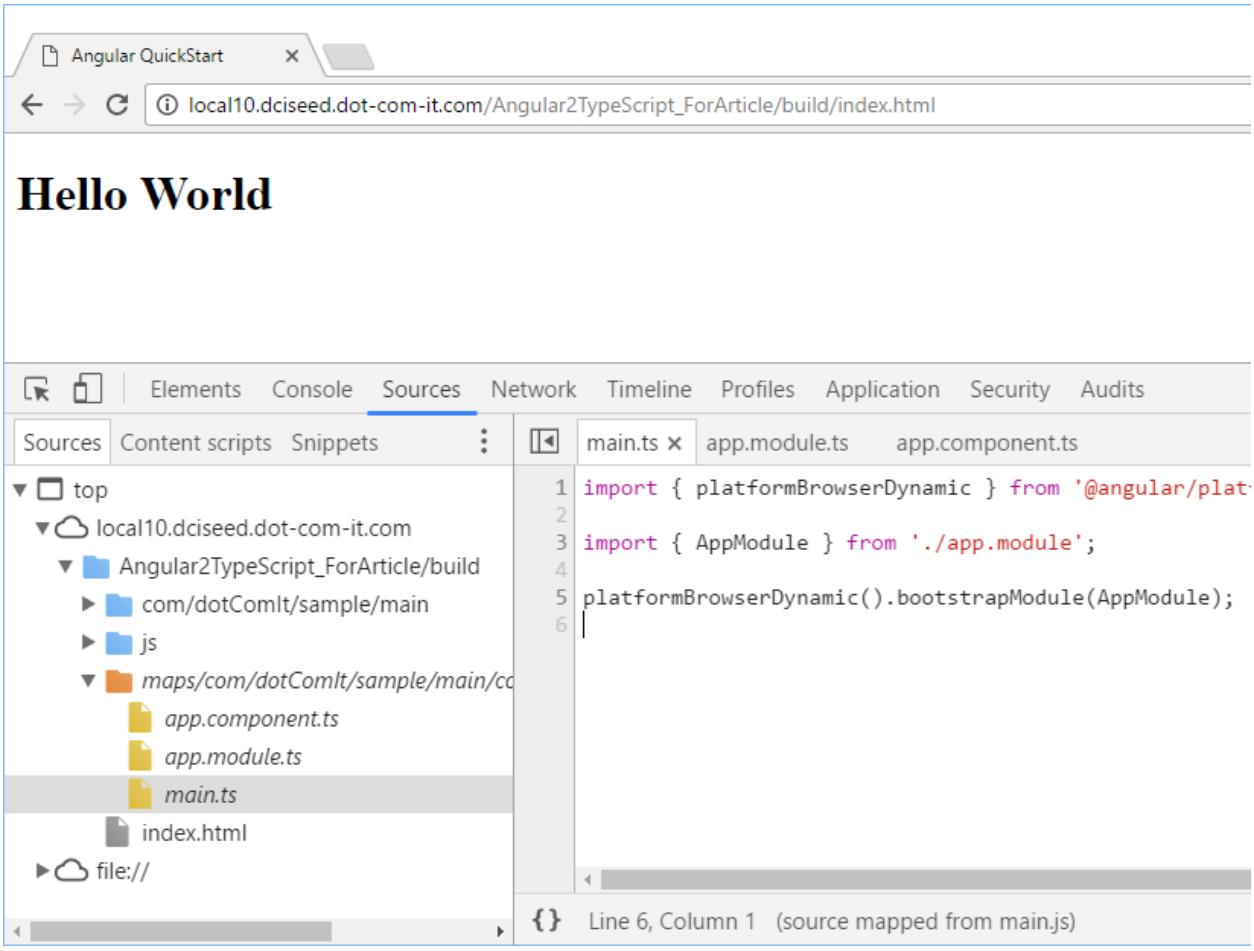


Check the build directory:



A source map was generated for each TypeScript source file. Success!

Open up the app in a browser, and bring up the source in your dev tools:



The chrome dev tools let us drill down into the TypeScript files, just as we had written them. They support watches and breakpoints, just as if the browser was running them directly. This is a very useful debug tool.

Minimizing JavaScript with UglifyJS

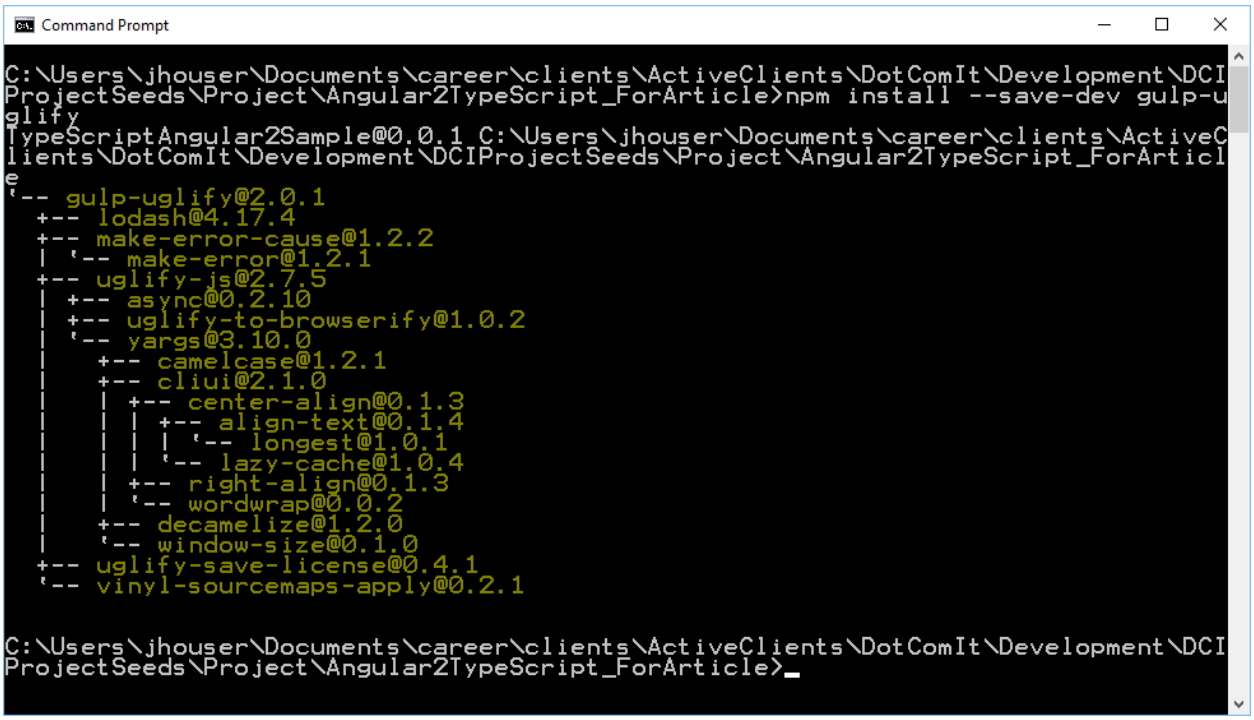
An important aspect of modern HTML5 development is to optimize your JavaScript files so they are as small as possible. The minification process shortens variable names, removes whitespace, and deletes comments. The purpose is to provide a smaller download to the end user. The process can, sometimes, be significantly especially with larger applications. UglifyJS is my preferred minimizer, so we'll use that.

Install gulp-uglify

The first step is to install the gulp-uglify module. Run this command:

```
npm install --save-dev gulp-uglify
```

You'll see feedback like this:



We are ready to use it in our script.

Modify Gulp Script

First, load the gulp-uglify script in the gulpfile.js:

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

Now, jump to the buildTS task:

```
gulp.task("buildTS", ["tslint"], function() {
    return gulp.src(typeScriptSource)
        .pipe(sourcemaps.init())
        .pipe(tsProject())
        .pipe(sourcemaps.write(mapPath))
```

```
        .pipe(gulp.dest(destinationPath));
});
```

We want to run Uglify before the source map is written, but after the TypeScript is converted. Add a single line:

```
return gulp.src(typeScriptSource)
    .pipe(sourcemaps.init())
    .pipe(tsProject())
    .pipe(uglify())
    .pipe(sourcemaps.write(mapPath))
    .pipe(gulp.dest(destinationPath));
```

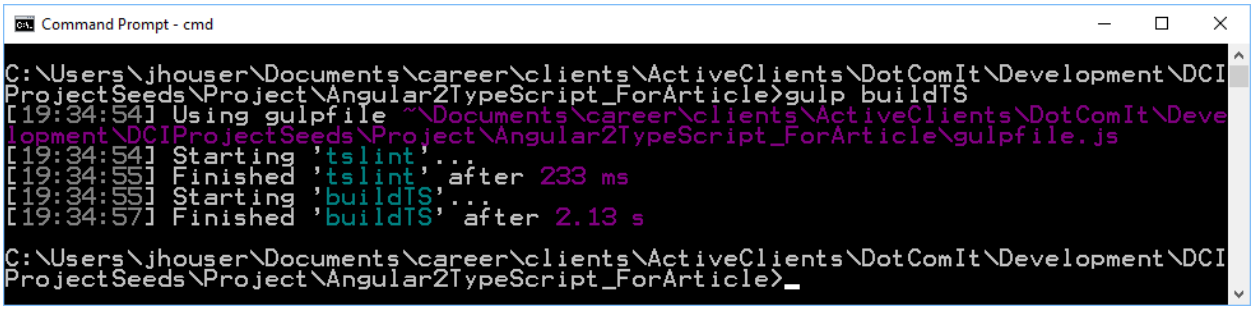
The pipe() after tsProject() calls the uglify() method. We could send in an object if we needed to configure something, but I have found things work acceptably without doing so.

Review the Minimized Code

Run the updated script:

```
gulp buildTS
```

See it run:



The directory structure will not have changed, but the contents of the custom JS files have. Take a look at the app.module.js:

```
"use strict";var __decorate=this&&this.__decorate||function(e,o,r,t){var
p,n=arguments.length,c=n<3?o:null===t?t=Object.getOwnPropertyDescriptor(o,r):
t;if("object"==typeof Reflect&&"function"==typeof
Reflect.decorate)c=Reflect.decorate(e,o,r,t);else for(var a=e.length-
1;a>=0;a--)(p=e[a])&&(c=(n<3?p(c):n>3?p(o,r,c):p(o,r))||c);return
n>3&&c&&Object.defineProperty(o,r,c),c},core_1=require("@angular/core"),platf
orm_browser_1=require("@angular/platform-
browser"),app_component_1=require("./app.component"),AppModule=function(){fun
ction e(){}return
e}();AppModule=__decorate([core_1.NgModule({imports:[platform_browser_1.Brows
erModule],declarations:[app_component_1.AppComponent],bootstrap:[app_componen
t_1.AppComponent]})],AppModule),exports.AppModule=AppModule;
```

This file is the minimized version of the translated TypeScript code. It includes a lot of the SystemJS configuration that we didn't have to write manually. If you look closely, you see a lot of the function arguments are changed into single character values. White space and line breaks are removed. Other optimizations can be made by the uglify library, such as variable definition optimizations. Such things are not present in the existing code.

Try the code in the browser, and you'll find it still runs and the source maps still work.

Create Different Build Options

We've build a lot of individual tasks; but sometimes you just want to run them all at once. This section will go over a lot of those options.

A Simple Build Task

The first task is a simple build task. It will just run all the other tasks from a single command. This is easy to put together:

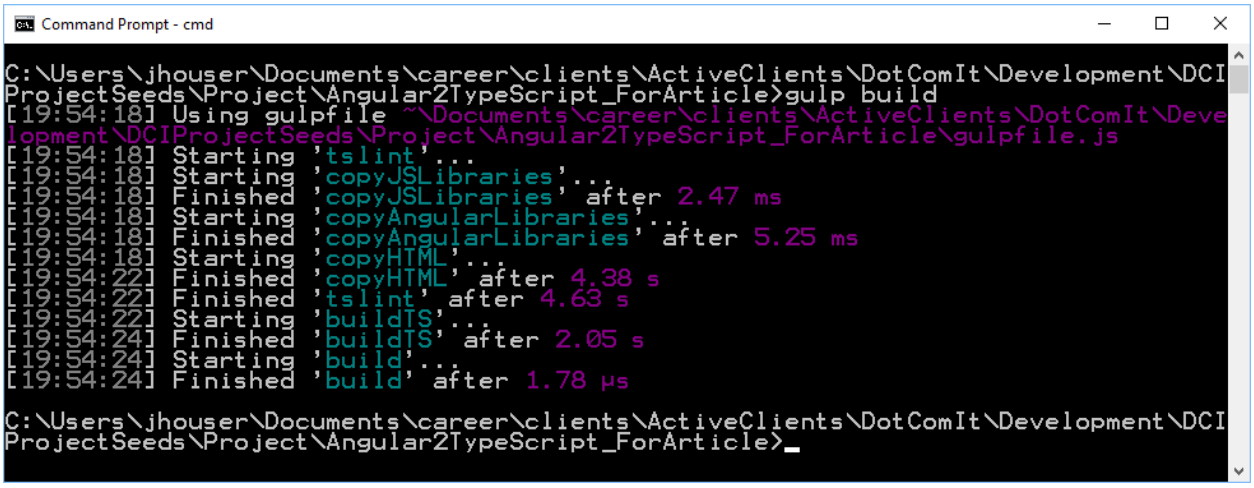
```
gulp.task("build", ['buildTS', 'copyJSLibraries',
                    'copyAngularLibraries','copyHTML']);
```

This creates a new Gulp task named build. The argument to the task is an array and each element of the array is a string which represents another Gulp task. We saw this approach with buildTS earlier, where the tslint task was executed before the actual buildTS task. In this case, the build task does not have its own functionality it just combines together the existing tasks.

Run this task:

```
gulp build
```

You'll see something like this:



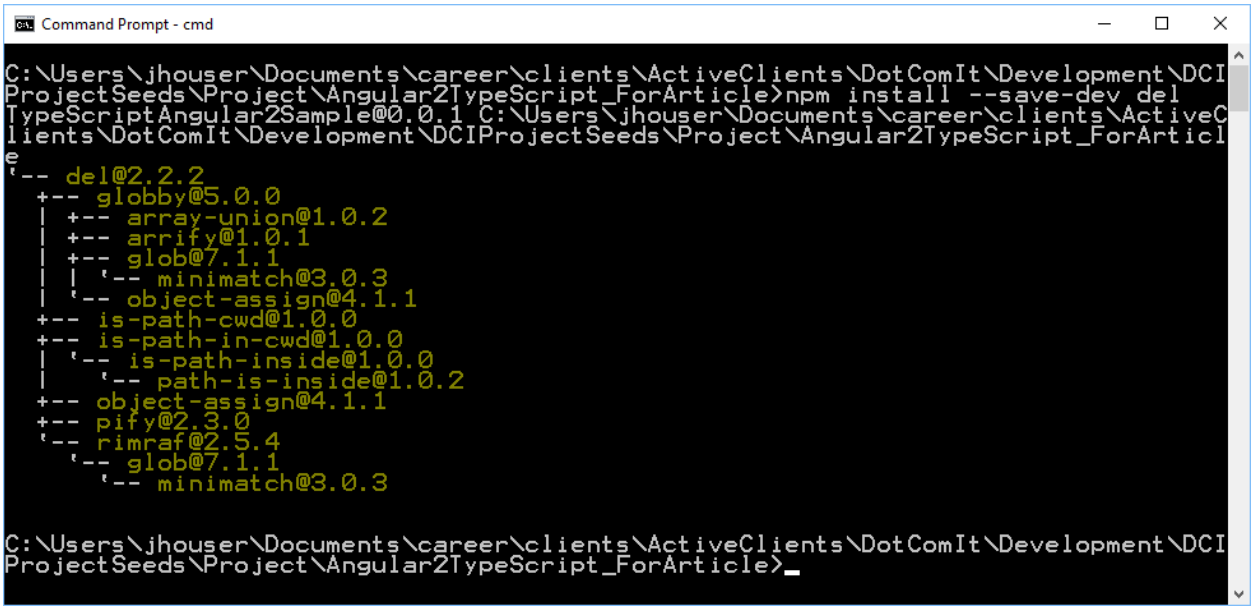
All the tasks are run, creating a build.

Delete the Build Directory

Sometimes you'll want to delete the build directory. To do that, we can use the NodeJS Plugin, del. This isn't a Gulp plugin, but can be wrapped in a gulp task. First, install it:

```
npm install --save-dev del
```

You'll see the install screen, like this:



Next, you can load the del library inside your gulpfile.js:

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

Create a glob array with everything that needs to be deleted:

```
var deletePath = [destinationPath + '/**']
```

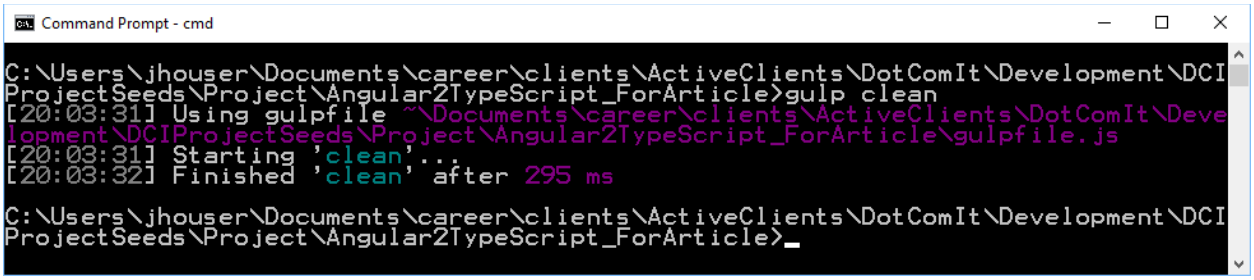
The destinationPath variable is used, with a wild card after it. This tells the del task to delete everything. Next, create the task:

```
gulp.task('clean', function () {
    return del(deletePath);
});
```

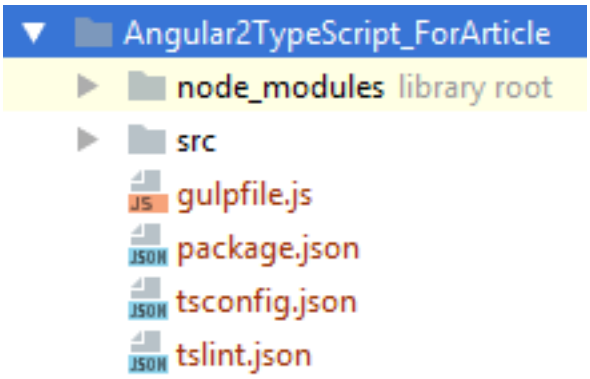
The task is named clean. It calls the del() module with the deletePath value. Run the task:

```
gulp clean
```

You'll see this:



Check your project directory:



The build directory is noticeably absent.

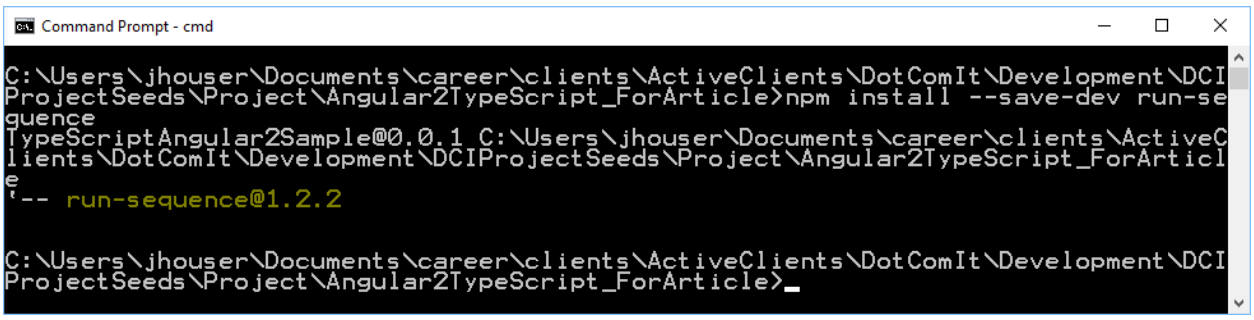
Create a Clean Build

Let's combine the clean task with the build path. To do that we'll want to run the two tasks in sequence, as we don't want the clean task to delete files the build task is creating. To do that we'll use a gulp plugin named run-sequence.

Install the plugin:

```
npm install --save-dev run-sequence
```

You'll see this:



With it installed, we can create an instance of it in the gulpfile.js:

```
var runSequence = require('run-sequence');
```

Then, create the task:

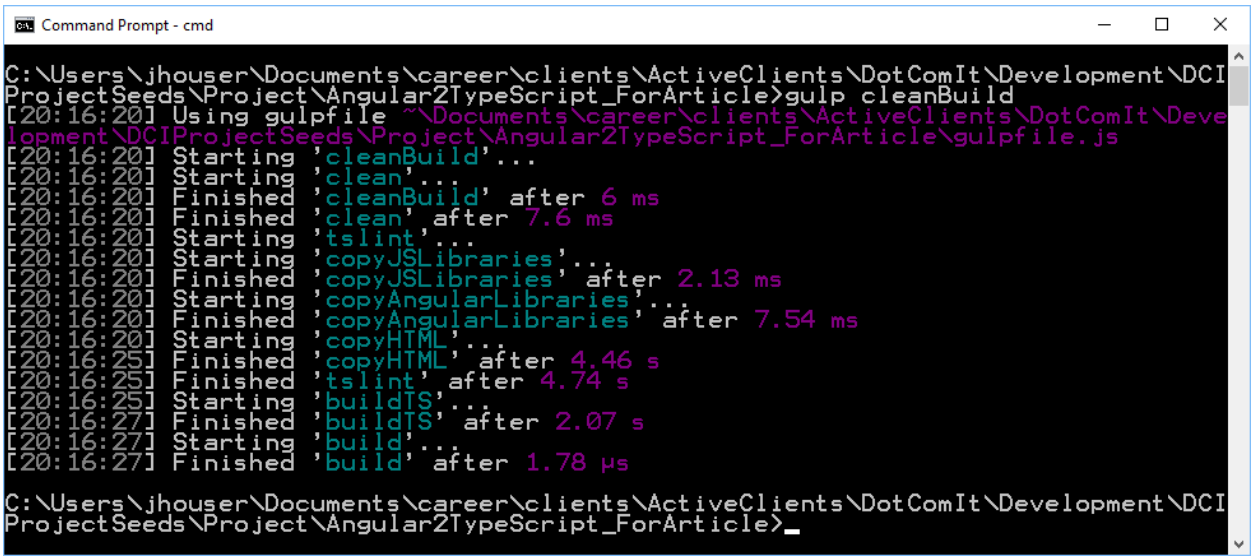
```
gulp.task('cleanBuild', function () {
    runSequence('clean', 'build');
});
```

I named the gulp task, cleanBuild. It uses the runSequence library to run the clean task—which deletes everything in the build directory, and the build task—which will create a fresh build.

Run the task:

```
gulp cleanBuild
```

You'll see something like this:



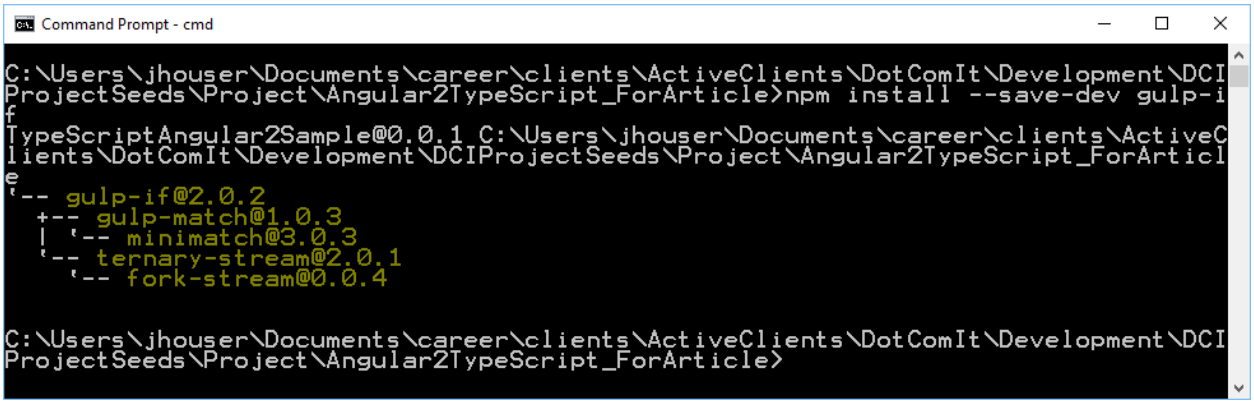
You see that the clean task is run first; and after it is finished the build tasks start. This will delete the build directory and all other relevant files, and then re-generate using the other tasks we wrote about earlier in this chapter.

Create a Production Build

Sometimes I like to create a build intended for production servers which does not include source maps. To make that happen, I use the gulp-if plugin to conditionally generate the source maps, or not. First, install gulp-if:

```
npm install --save-dev gulp-if
```

You'll see an install screen like this:



Now, import the gulp-if library as part of the gulpfile.js:

```
var gulpIf = require('gulp-if');
```

Before we modify the buildTS task, let's add a variable named devMode:

```
var devMode = true;
```

This is the variable we will use to determine whether or not to generate the source maps. It is set to true by default. Primarily we will change this variable as part of tasks, not as a configuration option.

Review the buildTS task:

```
gulp.task("buildTS", ["tslint"], function() {
    return gulp.src(typeScriptSource)
        .pipe(sourcemaps.init())
        .pipe(tsProject())
        .pipe(uglify())
        .pipe(sourcemaps.write(mapPath))
        .pipe(gulp.dest(destinationPath));
});
```

We want to use gulp-if as part of the two source map statements. First replace the source map init statement:

```
.pipe(gulpIf(devMode,sourcemaps.init()))
```

Instead of just calling sourcemaps.init(), we now wrap it in a gulpIf. This will check the devMode variable and conditionally init the source maps.

Also change the sourcemaps.write() pipe:

```
.pipe(gulpIf(devMode,sourcemaps.write(mapPath)))
```

With the buildTS task updated, we can now create a task for building a production version of the app. The purpose of this task is to set the devMode value to false; and then run the cleanBuild task:

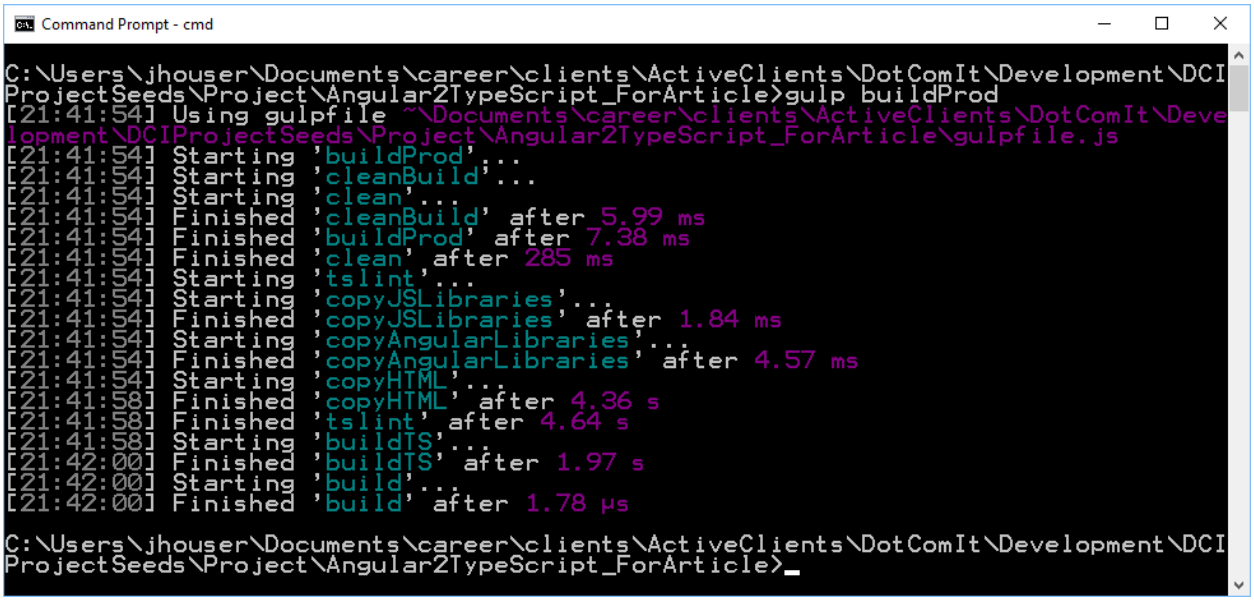
```
gulp.task('buildProd', function(){
    devMode = false;
    gulp.start('cleanBuild')
});
```

We can use gulp.start() to run the cleanBuild task. Running cleanBuild will delete the build directory, and then run the build task to compile the TypeScript files, move the HTML, and move the JavaScript libraries.

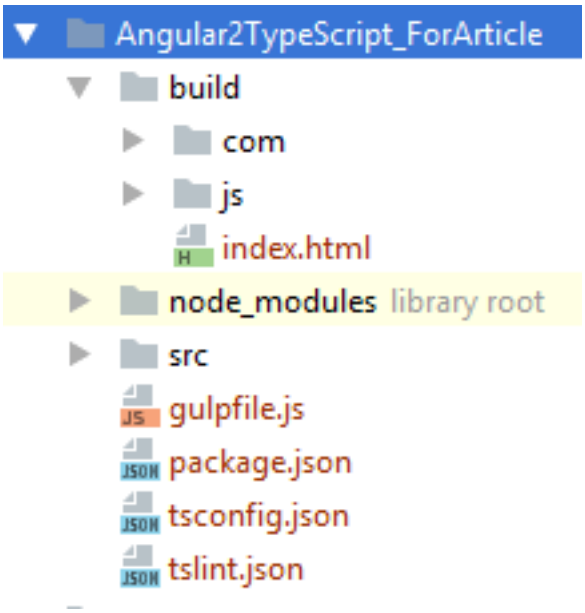
Run the task:

```
gulp buildProd
```

You should see this:



Take a look at the build directory:



You'll notice that the maps directory is missing; meaning that the sourcemaps were successfully bypassed when running the cleanBuild.

Watch the Directories for Code Changes

There was one more thing I wanted to touch on for the purposes of this article. When changing code, I want the app to automatically be recompiled. Gulp provides built in functionality to make that happen. Iterative builds help improve performance, and get you reviewing your app in the browser without having to manually compile every time.

Create a task named buildWatch:

```
gulp.task('buildWatch', ['build'], function(){
}
```

The first thing this task does is build the application. The array before the function does this. This makes sure that we have the most recent code in the build directory before we start making changes.

Then we need to start watching for changes. We can use the watch() method of gulp to do this. First, the TypeScript files:

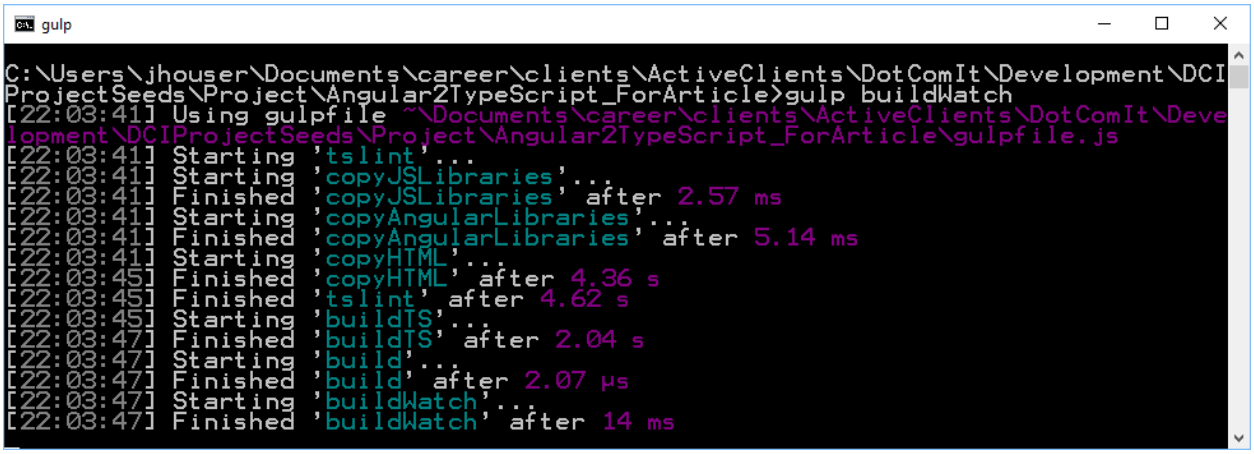
```
gulp.watch(typeScriptSource,['buildTS'])
    .on('change', function(event){
        console.log('File Path' + event.path);
})
```

The first argument to the watch() function is the glob array that points to the TypeScript source. The second argument is an array which represents the tasks to execute when a change is detected. In this case, the buildTS task is executed. When a change is detected, I chained an on() event after the watch() task. This outputs the name of the file that changed.

Let's try this:

```
gulp buildWatch
```

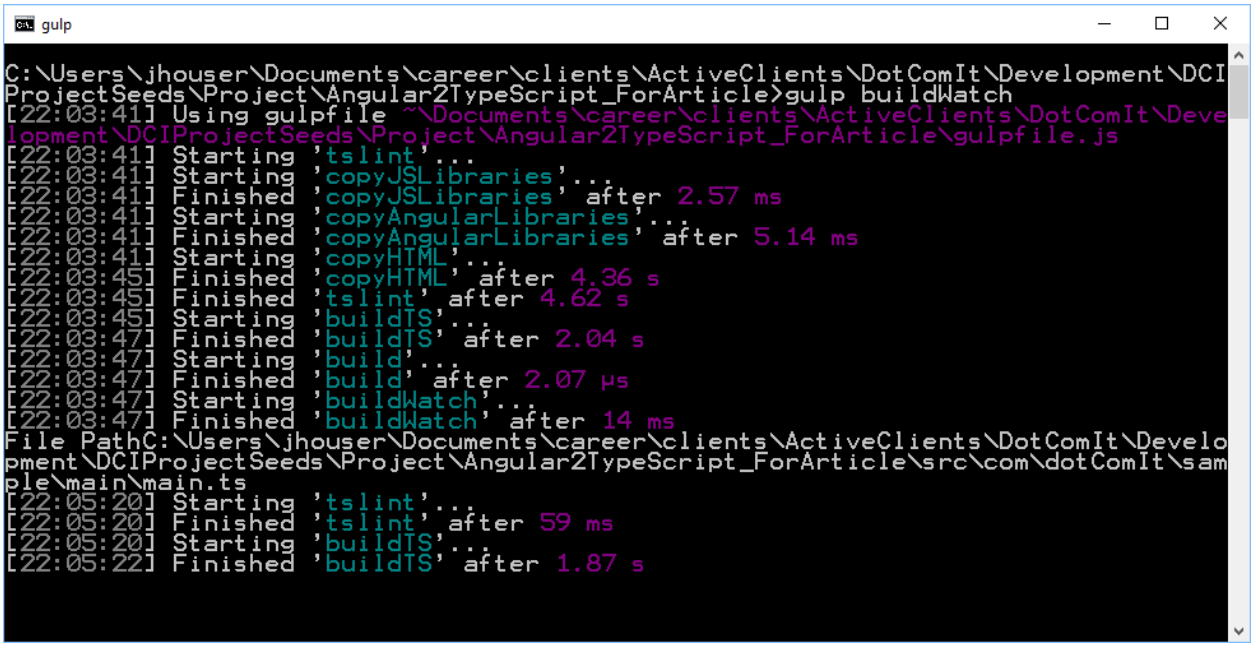
You should see something like this:



Notice that you are not sent back to the console prompt after running this. The task is waiting for something to happen. Change the main.ts file by adding something simple like this to the end:

```
console.log('something');
```

Then look at the console:



The changed file was output the console, and the buildTS was re-run. The buildTS task runs tslint before building the code; that has not changed.

You can use the exact same approach for watching changes with HTML and the JS Libraries:

```
gulp.watch(htmlSource,['copyHTML']).on('change', function(event){
    console.log('File Path' + event.path);
})
gulp.watch(javaScriptLibraries,['copyJSLibraries']).on('change',
function(event){
    console.log('File Path' + event.path);
})
```

As files are changed, the relevant tasks are rerun. I don't usually watch the Angular libraries because they are rarely changed or updated during development unless it is a big project decision.

Final Thoughts

Build scripts are important. When I started doing Angular 1 development with JavaScript, I could just use JavaScript directly in the browser and was able to push off dealing with build scripts to a future time. However, when using Angular 2 with TypeScript, the build process became much more important since a compile process is needed before you can test your code.

I hope you learned a lot and this is helpful during your Angular 2 adventures. Let me know how it helped you: jeffry@dot-com-it.com .