Building HTML5 Applications with TypeScript and Gulp

By Jeffry Houser

A DotComIt Whitepaper

Contents

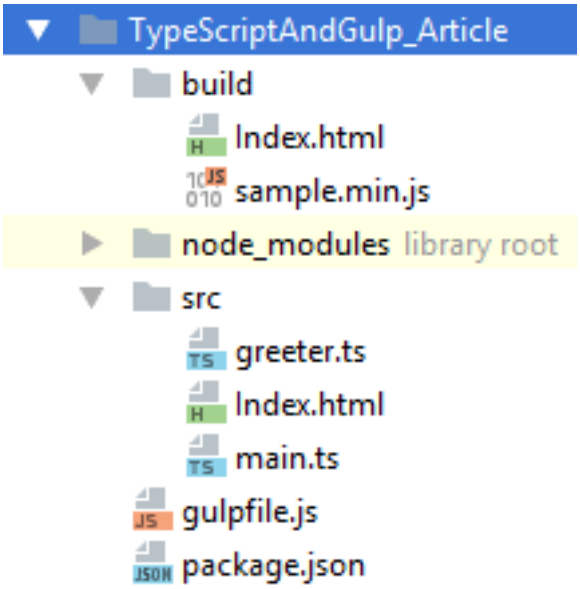
TypeScript is a newish language for application development on the web. It is a superset of JavaScript, and supports things like static typing and class based object oriented programming. Static typing, particularly, can offer improved tooling over what is currently available for simple JavaScript programming.

TypeScript can be used for any web application, or even on the server side with NodeJS. It is the recommended language for Angular 2 applications. TypeScript is compiled to regular JavaScript for deployment and testing. This white paper will explain the infrastructure we use to make this happen.

The TypeScript Application

Before we start to look at the build process, let's look at a super simple TypeScript application. We'll need some code to compile. This code borrows heavily from the official TypeScript tutorials. It will add a 'hello world' message to the body of an HTML page.

Create a directory for this setup. The final directory setup will look something like this:



This is a description of each directory:

- **TypeScriptAndGulp_Article**: The root directory will contain the script files for running Gulp and config file for NodeJS.
    - **build**: This directory will contain the final, processed, build.
    - **node_modules**: This directory will contain the NodeJS package, such as Gulp and TypeScript, to needed to perform the actions.
    - **src**: This directory will contain the application's source files.

We will create most of these files and directories as we move through the process.

For now, I named the project directory TypeScriptAndGulp_Article. Start by creating the src directory. I always separate the source code, compiled code, build scripts, and NodeJS modules. Inside the src directory, create an index.html file:

```
<html>
<head>
 <title>TypeScript Sample for DotComIt Article</title>
</head>
<body>
<script src="sample.min.js"></script>
</body>
</html>
```

This is a simple index.html file. There is one item I want to draw attention to. The script tag which imports a JavaScript file named sample.min.js. This is the file we will generate from the TypeScript code.

Next, create a file named greeter.ts. The ts extension is the standard extension for TypeScript files, similar to how 'js' is the standard extension for JavaScript files. For this sample I'll put all the files inside the root of the src directory, however in a more complex application, I may categorize the files under a descriptive directory structure.

This is the greeter.ts:

```
export function sayHello(name: string) {
    return `Hello from ${name}`;
}
```

This file exports a function named sayHello(). The export is not something you'll commonly see in browser JavaScript, but is used when creating NodeJS Modules. It, basically, says that "This is an API I Want people to use to access code in this file." The sayHello() method has a single parameter, name. The type of the name is specified, something that doesn't happen in JavaScript but adding it allows for type checking at the compiler level. The body of the method returns a string, referencing the name variable inside the string. The syntax is very similar to JavaScript, so learning TypeScript should not be a big hurdle for you to climb.

Now create a file named main.ts. This is the entry point to our application. Back in my school days we'd call the routine that ran the app a main routine, so this is a similar approach. The first thing the main.ts does is import the greeter file:

```
import { sayHello } from "./greeter";
```

This makes all the code from the greeter.ts file available to use in the main.ts file. Now, create a showHello() method:

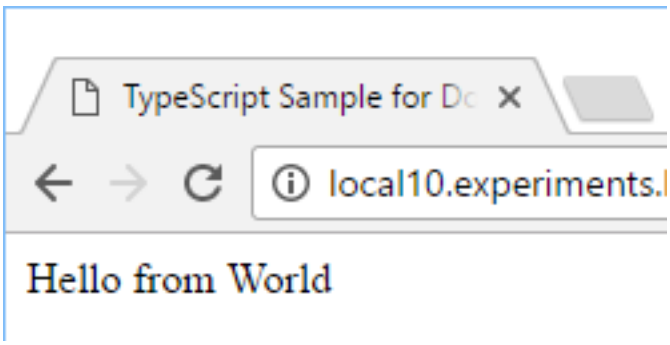
```
function showHello(name: string) {
    document.body.innerHTML = sayHello(name);
}
```

This method looks a lot like a JavaScript method definition, the only difference is that the name variable is given a type of string. The code accesses the innerHTML of the web pages body tags and outputs the results of the sayHello method, which will just be a string that says hello from the given name.

Finally, call the showHello() method:

```
showHello("World");
```

That completes the super simple TypeScript application. I could write a lot about TypeScript, however I wanted this article to focus on the build process of turning TypeScript into a JavaScript web application. As such, I'll leave the JavaScript there. When you finally compile the application, it should look something like this in the browser:



A Basic Compile Script

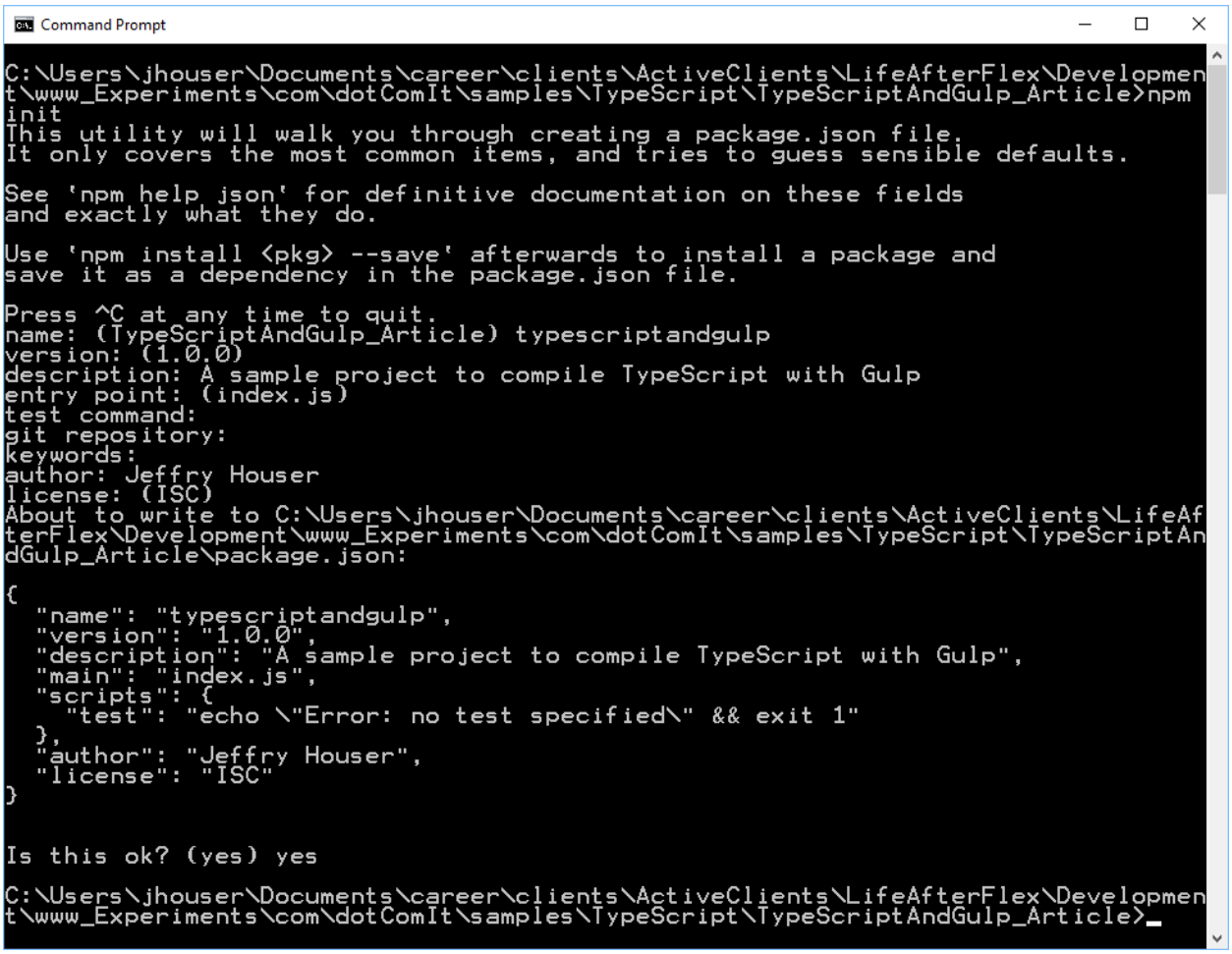
To compile TypeScript into JavaScript we are going to use NodeJS and a bunch of plugins.

Install and Setup NodeJS

The first step is to install NodeJS if you haven't already. The formal instructions will give you more detailed instructions if you haven't already done so. Then create a generic package.json configuration file. You can run this script:

```
npm init
```

And follow the instructions to create a simple package.json. You'll see something like this:



My final package.json is listed below:

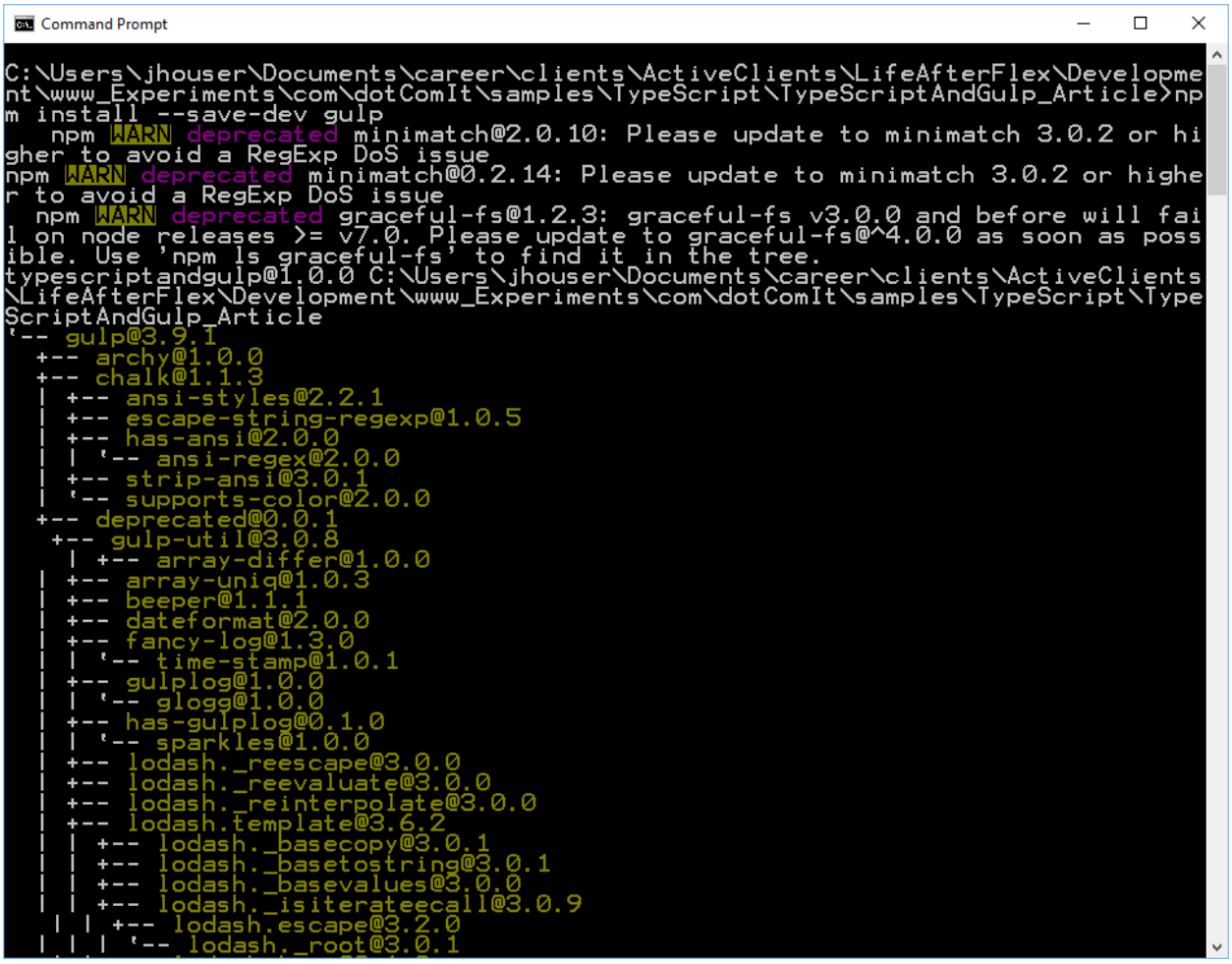
```
{
  "name": "typescriptandgulp",
  "version": "1.0.0",
  "description": "A sample project to compile TypeScript with Gulp",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Jeffry Houser",
  "license": "ISC",
  "repository": {}
}
```

Install Node Modules

Now we want to install some NodeJS plugins. First, we want to make sure Gulp is available. Gulp is the script runner we'll use to execute tasks to run our application. Run this command:

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

You'll see a screen similar to this:

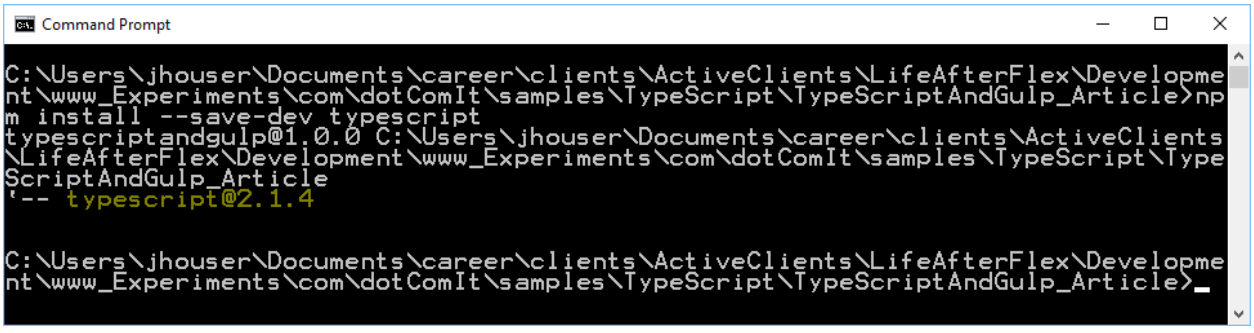


You can ignore the warnings. Your project directory will now have a node_modules directory.

Next, install the TypeScript plugin:

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

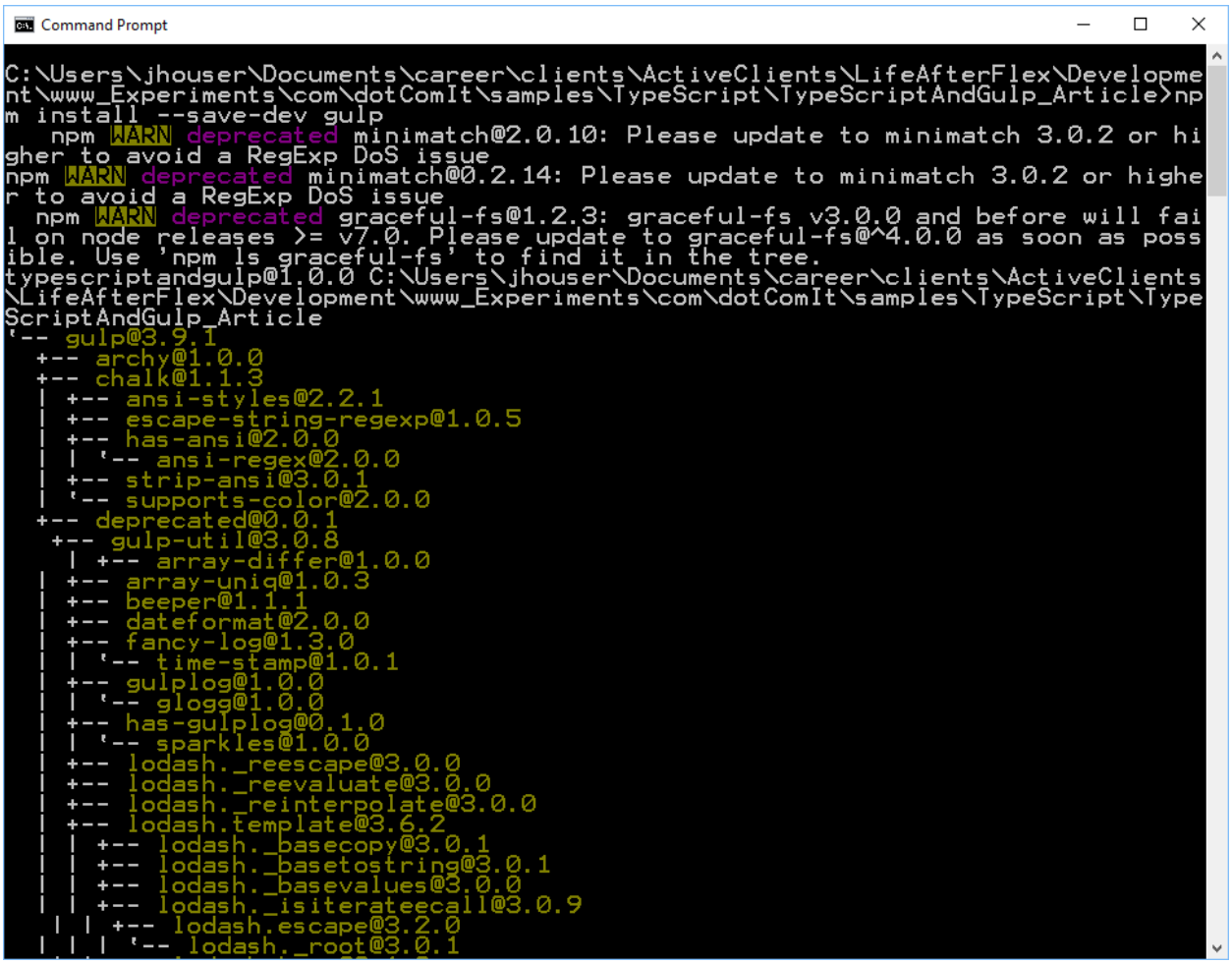
The NodeJS TypeScript plugin will not be used directly in the gulp script, but is a required dependency:



Next, install browserify. Browserify will combine all the JavaScript files into a single file. We also want tsify installed, a plugin that will allow browserify to compile TypeScript into JavaScript. First, install browserify:

```
npm install --save-dev browserify
```

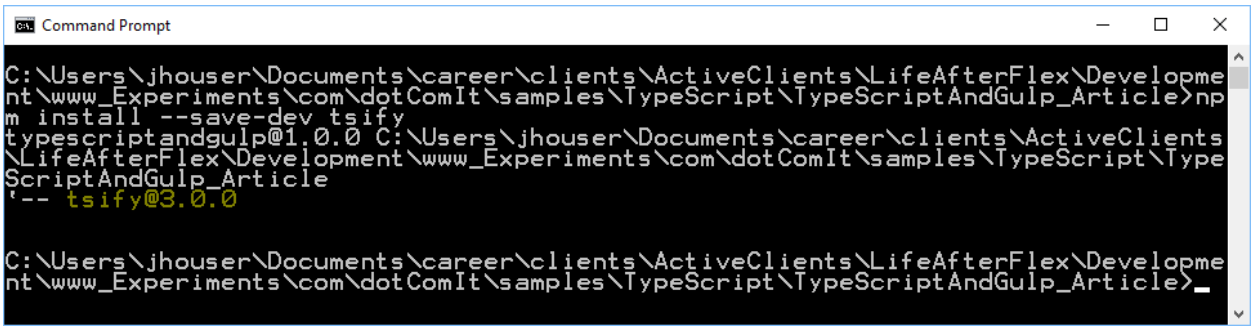
You'll see results like this:



Then install tsify:

```
npm install --save-dev tsify
```

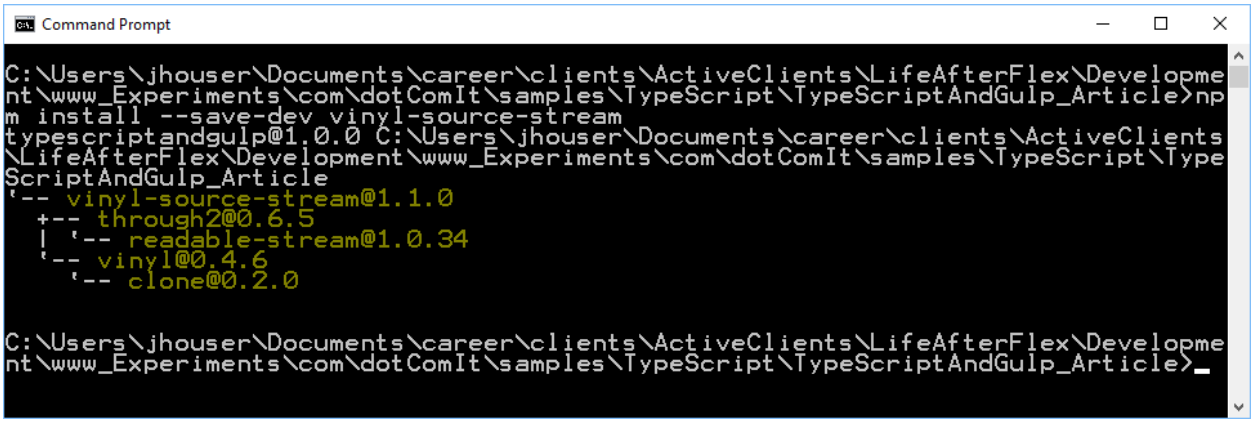
Your screen should look like this:



A Gulp stream object is named vinyl and is a virtual file format. To manipulate the output from Browserify with a Gulp script , we'll need to convert Browserify's object to the vinyl format. Thankfully there is a plugin for that, vinyl-source-stream module:

```
npm install --save-dev vinyl-source-stream
```

See the console results:



It is a lot of setup, but we should have all the required plugins we need to get started now.

Create the Gulp Task

In the root directory of the project, create a file named gulpfile.js. It is the default Gulp configuration file. First, load the gulp module:

```
var gulp = require("gulp");
```

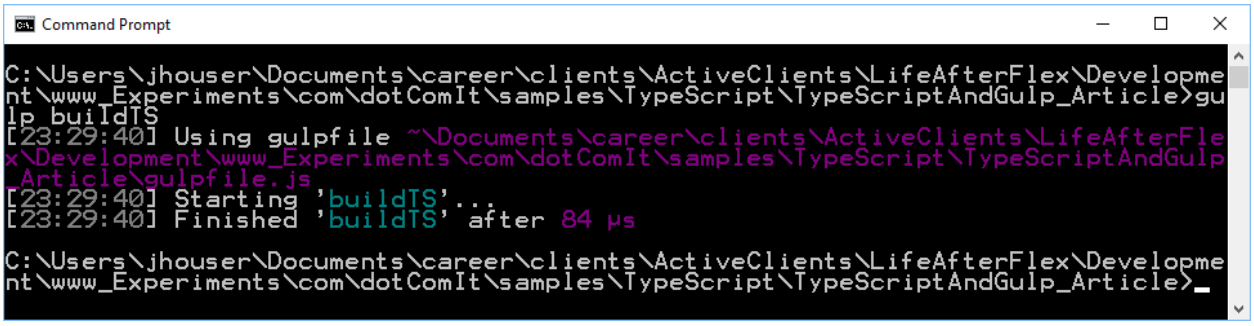
From there, you can create a gulp task for compiling TypeScript:

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

This is a blank task that does nothing. It calls the task() method on the gulp object. The task() method has two argument: the name of the task and a function to perform the task. You can execute this task by typing this into your console:

```
gulp buildTS
```

You should see something like this:



The task does nothing yet, though. Let's make it do something. Import the rest of the libraries:

```
var browserify = require("browserify");
var source = require('vinyl-source-stream');
var tsify = require("tsify");
```

Make sure these libraries are imported outside of the Gulp task. After the library import, still outside the Gulp task, let's define a few variables. The first is something I call appEntry:

```
var appEntries = ['src/main.ts']
```

It is an array that contains the main TypeScript file. Browserify will analyze main.ts to find all other files it uses and will do recursively. This is a bit different than the process I used in an AngularJS JavaScript application; where I'd process all files ending with the js extension.

Next, create a variable for the processed file name, and one for the final path:

```
var javaScriptDestinationFile = 'sample.min.js';
var destinationPath = 'build';
```

The destination file is the final output after the TypeScript files are merged and compiled into a single one. We referenced this file name in the index.html document. The second variable specifies the final destination of the file. I could define these as part of the Gulp task, but prefer to define them as global variables so they are easily accessible across multiple tasks, and I can tweak how everything works just by changing the config values.

Now go back inside the buildTS task and call the browserify() method:

```
return browserify({
    compilerOptions: {
        module: "commonjs",
        target: "es5",
    },
    entries:  appEntries
})
```

This calls the browserify() method and passes in an argument. The argument object specifies the configuration for this browserify process. Let's go through them:

- **compilerOptions**: This is another object that specifies how the TypeScript compiler will work.
    - **module**: Specifies how modules are loaded in UI code, in this case we are using the commonjs convention.
    - **target**: Specifies the ECMAScript version you want to output to; in this case I am specifying EcmaScript 5, however other versions are supported.
- **entries:** Specifies an array of entry points to your application.

This is a very simple config, and it can be a lot more complicated if you need it to be. Next, tell Browserify to compile the TypeScript code:

```
.plugin(tsify)
```

Each successive command is daisy chained on the other with the dot syntax. This uses the Browserify plugin() method to call the tsify package to compile the TypeScript code into JavaScript.

Next, call bundle() to merge all code into a single unit:

```
.bundle()
```

This one is simple. Then, specify the name of the new file:

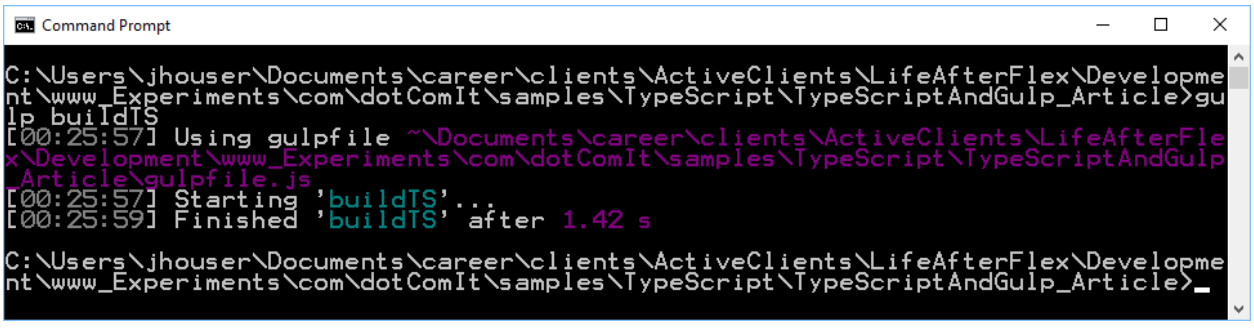
```
.pipe(source(javaScriptDestinationFile))
```

This uses the vinyl-source-stream module to turn the javaScriptDestinationFile name into something that can be accessed via Gulp. The pipe() method means that we are getting ready to output information, instead of processing input. Finally, specify the destination of the new file:

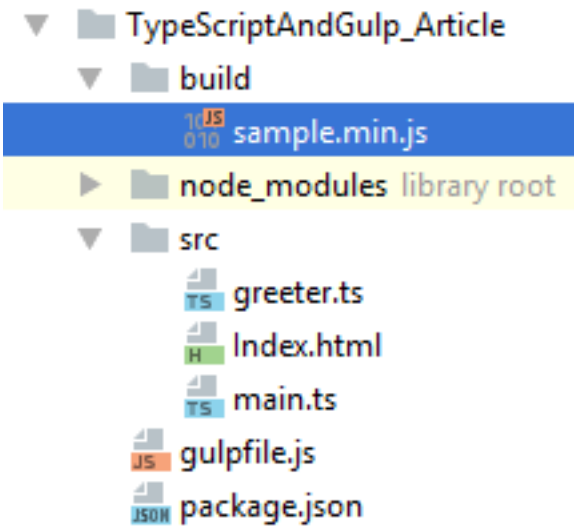
```
.pipe(gulp.dest(destinationPath));
```

This uses the pipe() operator too; and the value is Gulp's dest() function. This, basically, tells the script to put all our files in the build directory.

You can run the gulp script:



Look at the directory structure after our first run:



All the node modules we installed are in the node_modules directory. The source we looked at earlier is in the src directory. The JavaScript file built from the TypeScript files is in the build directory. The gulpfile.js and package.json are in the root application directory. I like this structure because it cleanly separates concerns.

Things are not yet runnable in a browser, because the Index.html was not moved from the src directory to the build directory. Before we work on copying the html file; let's review the compiled sample.min.js.

Review the compiled TypeScript

Let's take a look at the JavaScript file that was created from our compile process:

```
(function e(t,n,r){function s(o,u){if(!n[o]){if(!t[o]){var a=typeof
require=="function"&&require;if(!u&&a)return a(o,!0);if(i)return i(o,!0);var
f=new Error("Cannot find module '"+o+"'");throw
f.code="MODULE_NOT_FOUND",f}var
l=n[o]={exports:{}};t[o][0].call(l.exports,function(e){var
n=t[o][1][e];return s(n?n:e)},l,l.exports,e,t,n,r)}return n[o].exports}var
i=typeof require=="function"&&require;for(var
o=0;o<r.length;o++)s(r[o]);return s})({1:[function(require,module,exports){

"use strict";
function sayHello(name) {
    return "Hello from " + name;
}
exports.sayHello = sayHello;

},{}],2:[function(require,module,exports){

"use strict";
var greeter_1 = require("./greeter");
function showHello(name) {
    document.body.innerHTML = greeter_1.sayHello(name);
}
```

```
showHello("World");

},{"./greeter":1}]},{},[2]);
```

There is a lot going on here. The export and import functionality is not native to JavaScript. Browserify automatically creates a module loader function to handle this, and that is the first function on the page. I'm not going to dissect that for the purposes of this white paper. The greeter module and main module are there. You'll notice the JavaScript code is a bit different than the main code.

Let's compare side by side:

Original Code	Compiled Code
`export function sayHello(name: string)` `{` `    return `Hello from ${name}`;` `}`	`"use strict";` `function sayHello(name)` `{` `    return "Hello from " + name;` `}` `exports.sayHello = sayHello;`

The compiled code adds a use strict command in front of the method. This tells the code to execute in strict mode, which forces all variables to be declared. Next is the function definition. The TypeScript specifies a variable name for the argument, whereas that JavaScript does not. JavaScript does not have types, so that is not needed. The function export was moved from the function definition to a line at the end of the code segment, which puts the sayHello() function inside an exports variable. Finally, the return statement of the sayHello() function moved the variable argument from inside the quotes to the outside of the quotes. The code is similar; it has just been modified to appropriately run in the browser.

Original Code	Compiled Code
`import { sayHello } from "./greeter";` `function showHello(name: string) {` `    document.body.innerHTML =` `sayHello(name);` `}` `showHello("World");`	`"use strict";` `var greeter_1 = require("./greeter");` `function showHello(name) {` `    document.body.innerHTML =` `greeter_1.sayHello(name);` `}` `showHello("World");`

This compiled code also uses the use strict command here, as it did above. The import statement is translated into a require() statement. Require is common in NodeJS, but is not built into browsers. The Browserify special code is what is required to support require in the browser. The function definition is the same, with the caveat that that the argument is no longer typed. The sayHello() method call is almost identical; but the compiled code references the module variable, greeter_1 instead of just calling the module's method. The final showHello() method call is unchanged.

This is a simple sample, but should give you a generic idea of what is going on behind the scenes. For most practical purposes you won't have to delve into the compiled code, or compare it to the original code.

Copy HTML files

We need to copy the HTML files from the source directory to the build directory. With our simple sample, the only HTML file is the index file; however a real web application may have multiple files, or many HTML template files that integrate with a framework such as Angular. We'll probably want to copy those from the src to the build directory also. We can use Gulp to easily copy files from one location to another.

First, create a global variable for the source:

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

I split this up into two variables, a sourceRoot that could be used for multiple paths as needed, and an htmlSource array. The array contains a single item, which specifies some wild cards that say look for all files that end in the html extension in the source root, and all of its subdirectories. While we're at it; we can modify the appEntries variable to make use of the sourceRoot variable instead of hard coding the src directory:

```
var appEntries = [sourceRoot + '/main.ts']
```

Now create the task to copy the HTML:

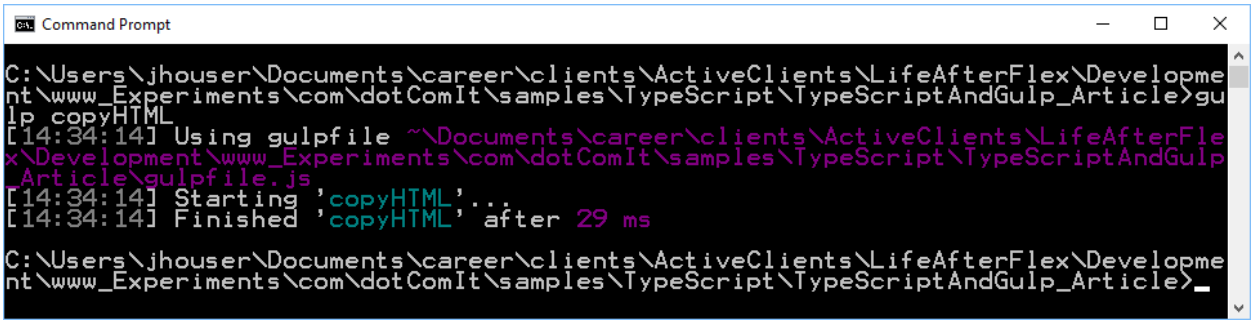
```
gulp.task('copyHTML', function () {
    return gulp.src(htmlSource)
        .pipe(gulp.dest(destinationPath));
});
```

The task is named copyHTML. We call the src() function on the gulp object. The src() function specifies the directories that this gulp task will process. Then the pipe() function is called to specify the next step in the process. In this case, we are not processing the files in anyway, just moving them from one place to another. So, the pipe() just uses gulp.dest() to specify the location of the new files. The path for the src files will be retained during the copy.

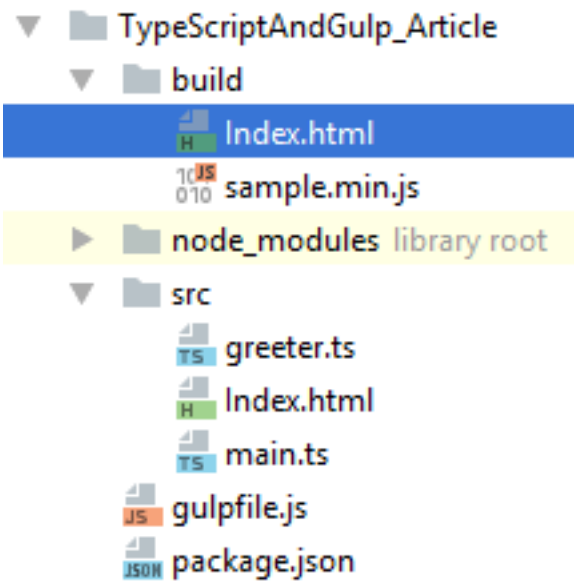
Run this command:

```
gulp buildHTML
```

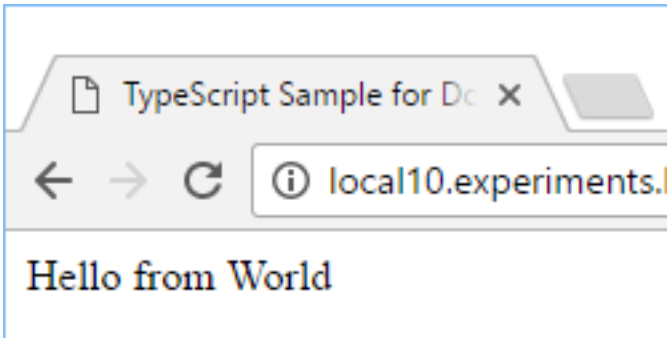
You should see something like this:



Not very descriptive output, but if you reexamine the directory structure, you'll see that the index.html file is now in the build directory:



Now we also have a build that can be run in the browser:



It isn't an interesting app, but it does prove that Gulp is successfully compiling TypeScript and copying the HTML files from the source directory into the build directory.

Perform Two Tasks in One Build

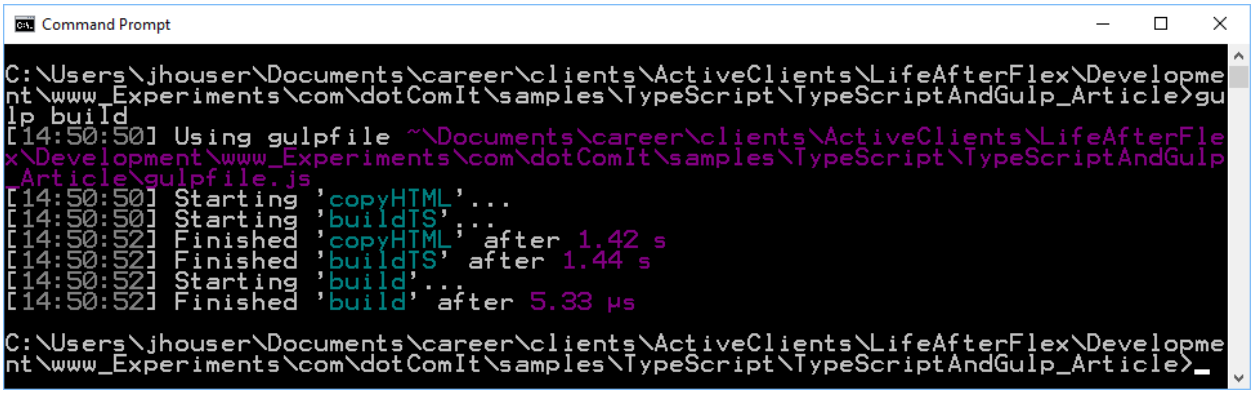
Gulp allows us to perform two tasks in a single build. When the gulp task is created, instead of specifying a function, we can specify an array of other tasks, like this:

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

The new task is named build. It doesn't contain any new functionality, just calls the copyHTML and buildTS tasks. Run it:

```
gulp build
```

You should see results like this:



If you check the build directory, you'll see that the HTML has been copied and the JavaScript file has been created.

Using Source Maps

When running your compiled code in a browser, it is difficult to tell where errors are happening with respect to your original code, because the code has actually changed. We can fix that by creating something called a source map. Browserify has source map support built right in.

Create Source Maps

First, create a global variable named devMode in gulpfile.js:

```
var devMode = true;
```

We can use this variable to turn on and off certain things, depending on whether we are building a dev build or a production build. For now we're just going to use it to enable or disable source maps, but I have used it for other reasons with different clients.

Inside the buildTS task, add the debug property to the browserify configuration object:

```
return browserify({
    compilerOptions: {
        module: "commonjs",
        target: "es5",
```

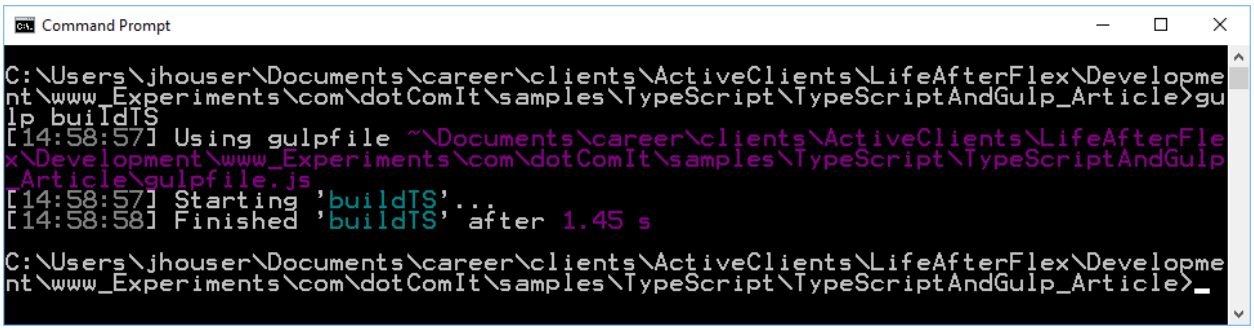
```
    },
    debug: devMode,
    entries:  appEntries
})
```

The value of the debug mode relates to the devMode variable. This property tells browserify to create source maps for the processed files.

Rerun the build:

```
gulp buildTS
```

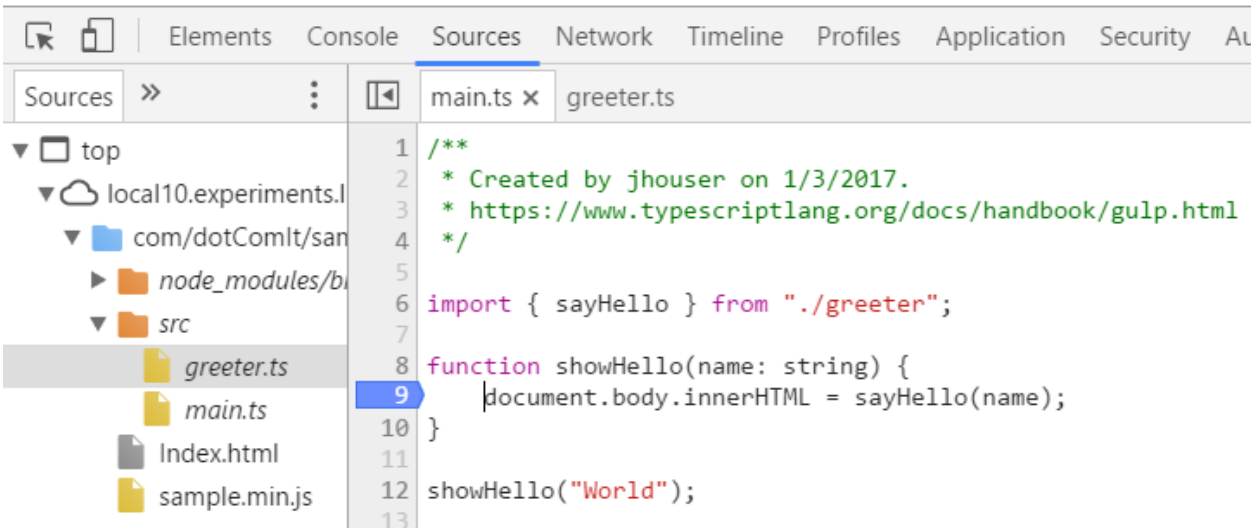
You'll see this screen:



The build feedback does not give any indication that something new occurred. Open up the sample.min.js file from the src directory and look at the bottom of the file:

```
//# sourceMappingURL=data:application/json;charset=utf-
8;base64,eyJ2ZXJzaW9uIjozLCJzb3VyY2VzIjpbIm5vZGVfbW9kdWxl
```

I truncated the response, but basically, you now see a sourceMappingURL comment at the bottom of the file, followed by a long encoded string. Your browser will know how to use this string to reconstruct your original code. Here you see the original code in the Chrome debug tools:



You can debug the source maps as if the browser was running that code explicitly, including adding debug points in the original code. This is a very useful debugging tool, but I do not like to generate source maps when creating a build intended for production.

Extract Source Maps to an External File

You may have noticed that the source map is generated as part of the sample.min.js file. This is functional however I prefer to keep source maps in an external file. Browserify does not support that inherently, however there is a plugin, named exorcist, that allows us to separate the source maps from the compiled JavaScript.

First, install exorcist:

```
npm install --save-dev exorcist
```

Unfortunately, at the time of this writing, the npm package is a step behind the master source. You can force NodeJS to install the most recent from the git repository:

```
npm i thlorenz/exorcist#master
```

You should see console results similar to this:

I needed to force the updated because the repository code had some changes which related to creating non-existent directories. I'm sure this is a temporary problem, and that the npm package will be updated soon.

Now that exorcist is installed, we can use it in the gulpfile.js. First, import it the exorcist module:

```
var exorcist    = require('exorcist')
```

Create a bunch of variables to clarify the final location of the maps:

```
var mapPath = 'maps';
var mapFile = 'sample.js.map'
var mapDestination = destinationPath + '/' + mapPath + '/' + mapFile;
var mapDestinationURL = mapPath + '/' + mapFile;
```

The mapPath variable specifies the relative location of the source maps. The mapFile variable specifies the name of the map file. The mapDestination is the full path to the map file. The mapDestinationURL is the relative URL to the map file. The first two values are helper values, but the second two will be used in the buildTS script when extracting the map files from the sample.min.js.

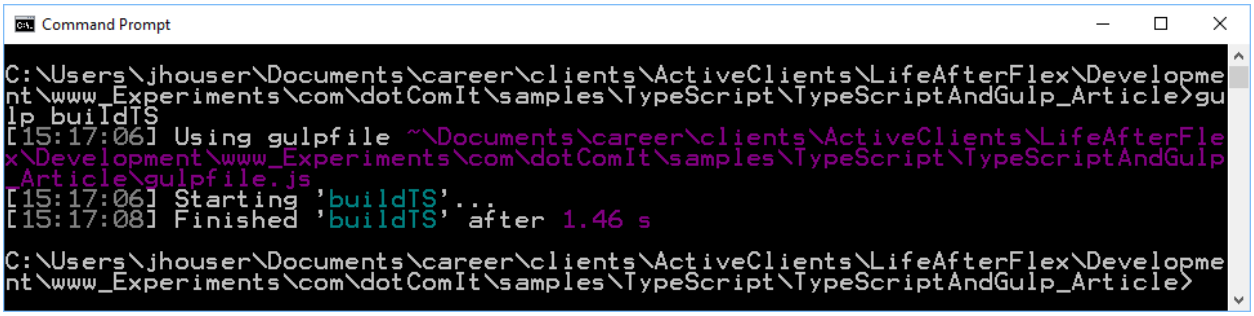
In the buildTS task, between the .bundle() call and the source() call, use exorcist:

```
.bundle()
.pipe(exorcist(mapDestination, mapDestinationURL))
.pipe(source(javaScriptDestinationFile))
```

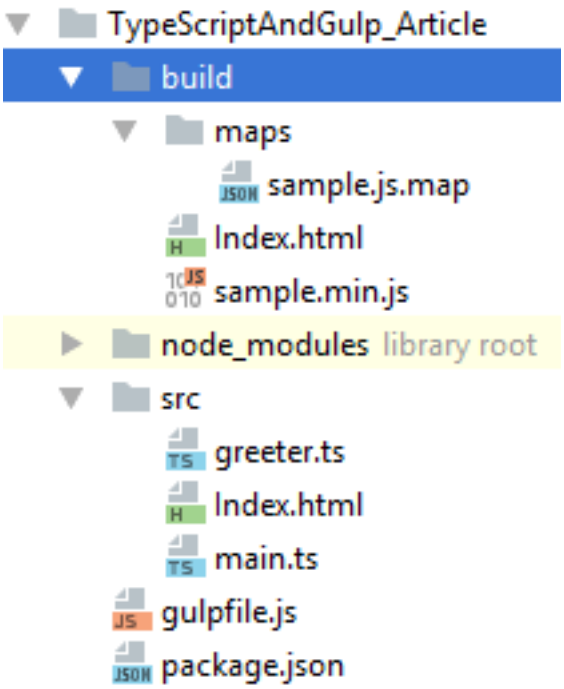
This is all we need to do. Rerun the task:

```
gulp buildTS
```

You should see results, like this:



Nothing new in the console, but review the directory structure:



You'll see the build directory now has a maps directory along with a sample.js.map directory. Open up the sample.min.js file and look at the bottom:

```
//# sourceMappingURL=maps/sample.js.map
```

The sourceMappingURL now specifies a URL instead of an inline template.

Open up the sample.js.map file and you'll see a JSON packet:

```
{
  "version": 3,
  "sources": [
    "node_modules/browser-pack/_prelude.js",
    "src/greeter.ts",
    "src/main.ts"
  ],
  "names": [],
  "mappings": "AAAA;ACAA; ",
  "file": "generated.js",
  "sourceRoot": "",
  "sourcesContent": [
    "(function e(t,n,r){function s(o,u){if(!n[o]){if(!t[o]){var a=typeof
require==\"function\"&& import { sayHello } from
\"./greeter\";\r\n\r\nfunction showHello(name: string) {\r\n
document.body.innerHTML =
sayHello(name);\r\n}\r\n\r\nshowHello(\"World\");\r\n"
  ]
}
```

I truncated this file for space reasons in the text here. This file tells the browser how to match sample.min.js to the original source. The sources property clarifies the files, and the sourcesContent specifies the content that should show up in those content files. Source maps are a powerful debugging tool.

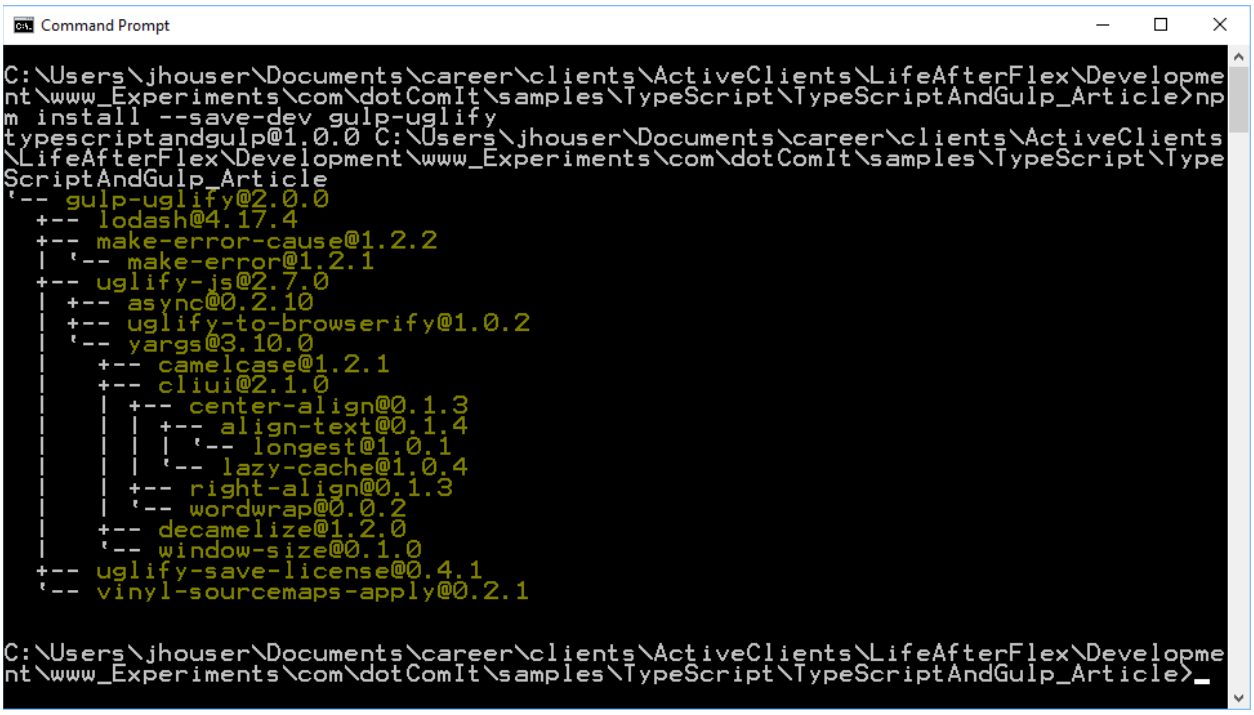
Minimizing JavaScript Code w/ Uglify

An important step in any JavaScript compilation process is minification. This is the process of shrinking your JavaScript file. Variable names are shortened, white space is removed. Sometimes the code is optimized to remove redundancies that make it easy to understand the code, but are not needed for perfect execution. The main reason for this is to give your users smaller downloads when a web browser loads the application. For small applications, the size difference may not matter, but with larger applications I have seen size reductions of over 50%; which is significant.

We are going to need a few gulp plugins to make this work. The first is gulp-uglify. UglifyJS is my preferred minimizer, and gulp-uglify is a gulp plugin for UglifyJS. Install it:

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

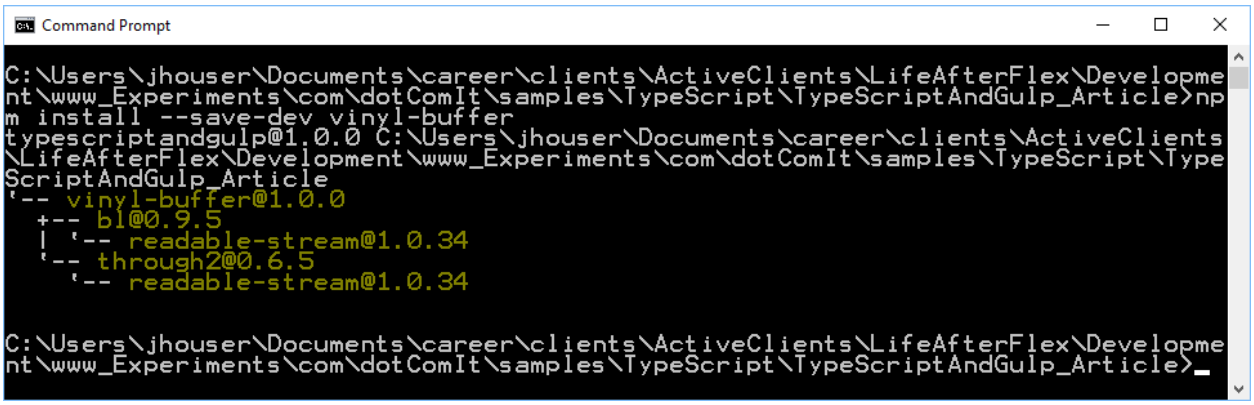
You should see a screen like this:



Next, we need vinyl-buffer. This is used to buffer the Browserify output so that a gulp plugin can easily manipulate it. Remember that vinyl is the name of the internal file format used by Gulp. Install it:

```
npm install --save-dev vinyl-buffer
```

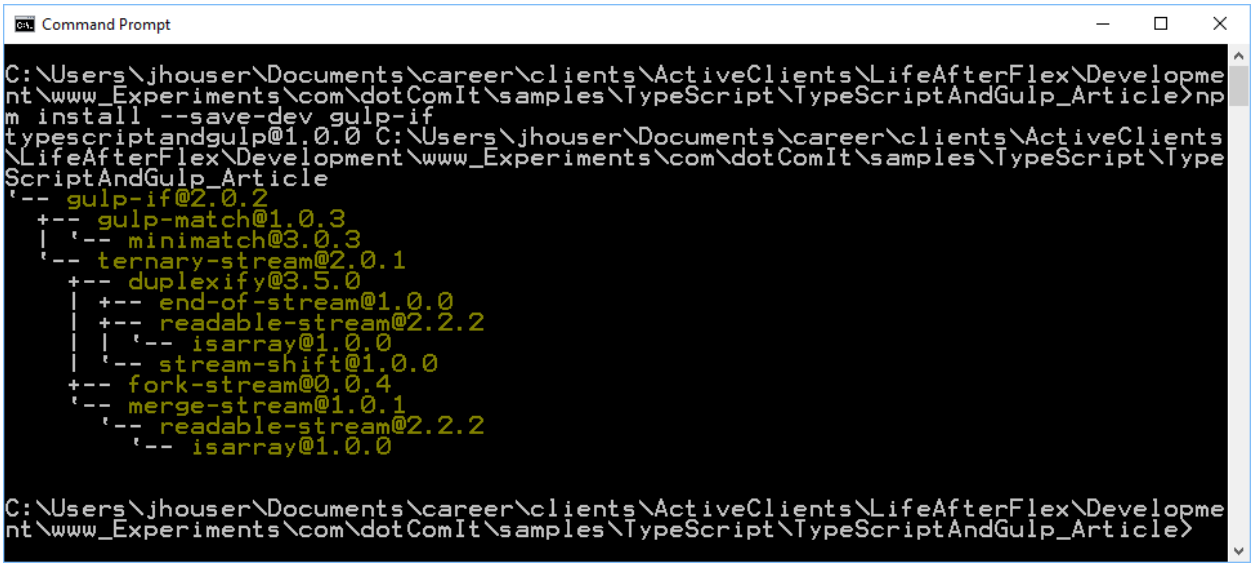
You'll see an install screen like this:



Finally, I want to install a plugin called Gulp-if. We'll use this to conditionally run the uglify plugin based on whether we are creating a dev build or a production build. Dev builds will not include uglify; but production builds will. Install it:

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

Run the command, and your console should look something like this:



Now, we can shift our attention back to the gulpfile.js. Import the three new modules:

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

Move down to the buildTS gulp script. After the source is specified, and before the destination, we use the buffer() command, and then gulpIf to run Uglify:

```
.pipe(source(javaScriptDestinationFile))
.pipe(buffer())
```

```
.pipe(gulpIf(!devMode,uglify()))
.pipe(gulp.dest(destinationPath));
```

The first line is the source, and we added earlier in the chapter. Next, the buffer() is called, which turns the vinyl file into a transform stream. Next, we use the gulpIf. If the devMode variable is false, then uglify will run. Otherwise nothing will happen. The final code in this segment saves the stream to a final destination—we already saw this code.

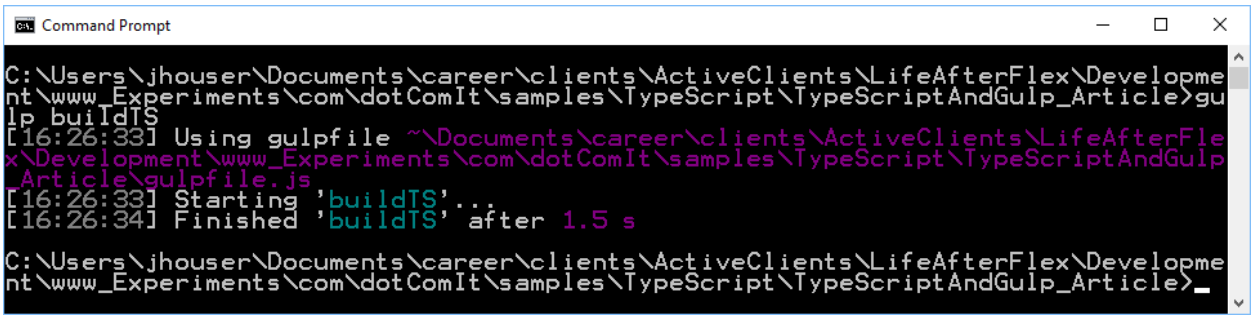
For demonstration purposes, swap the devMode variable to false:

```
var devMode = false;
```

Then run the script:

```
gulp buildTS
```

You should see output like this:



Open up the sample.min.js file from the build directory:

```
!function r(e,n,t){function
o(i,f){if(!n[i]){if(!e[i]){var c="function"==typeof
require&&require;if(!f&&c)return c(i,!0);if(u)return
u(i,!0);var l=new Error("Cannot find module
'"+i+"'");throw l.code="MODULE_NOT_FOUND",l}var
a=n[i]={exports:{}};e[i][0].call(a.exports,function(r){va
r n=e[i][1][r];return
o(n?n:r)},a,a.exports,r,e,n,t)}return
n[i].exports}for(var u="function"==typeof
require&&require,i=0;i<t.length;i++)o(t[i]);return
o}({1:[function(r,e,n){"use strict";function
t(r){return"Hello from
"+r}n.sayHello=t},{}],2:[function(r,e,n){"use
strict";function
t(r){document.body.innerHTML=o.sayHello(r)}var
o=r("./greeter");t("World")},{"./greeter":1}]},{},[2]);
```

It contains a lot of gobbly-gook. The variable names and function names were all minimized. Comments and white spaces were removed. Run the code in the browser, and find the app still works as expected. This process is important in presenting optimized HTML5 applications to your users.

Compile Code on the Fly with a Watch Script

Gulp allows us to automatically run a gulp task when a certain directory changes. This is built in and we don't need an additional plugin to make it happen. We have two tasks at the moment, one for copying HTML files and one for watching the TypeScript files. There is already a variable that represents the HTML source:

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

Let's create one for the TypeScript source:

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

The approach is the same, using wildcards to look at all subdirectories inside the sourceRoot folder.

Now create a gulp task called buildWatch:

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

You'll notice that there are three arguments to this gulp task, something we hadn't seen before. The first is the task name, buildWatch. The second is an array of strings with each string represents a gulp task. These tasks will run before the third argument's function is executed. Here we are only running one task, build, before starting the buildWatch task.

Here is the function code:

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

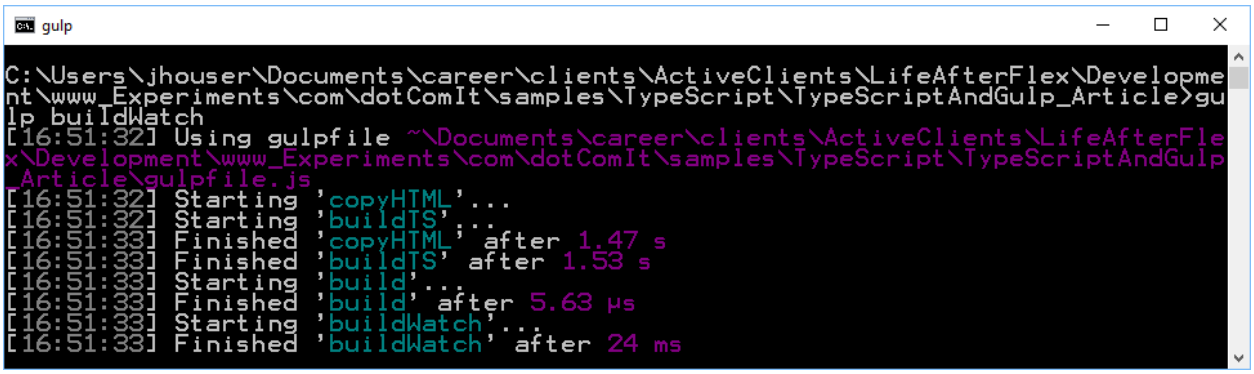
We use the watch() function on the Gulp object. It accepts two arguments. The first argument is a source array, in this case the htmlSource value. The second argument is a task array. In this case just the copyHTML task. Reading this in English, it says watch for files in the htmlSource, and when something changes run the copyHTML task. Dot notation is used to daisy chain an 'on' method to the watch. When the change event occurs, the type of event and modified files are output to the console.

The same syntax is used to watch for TypeScript changes. The watched path array is typeScriptSource instead of htmlSource. The task array contains buildTS instead of copyHTML.

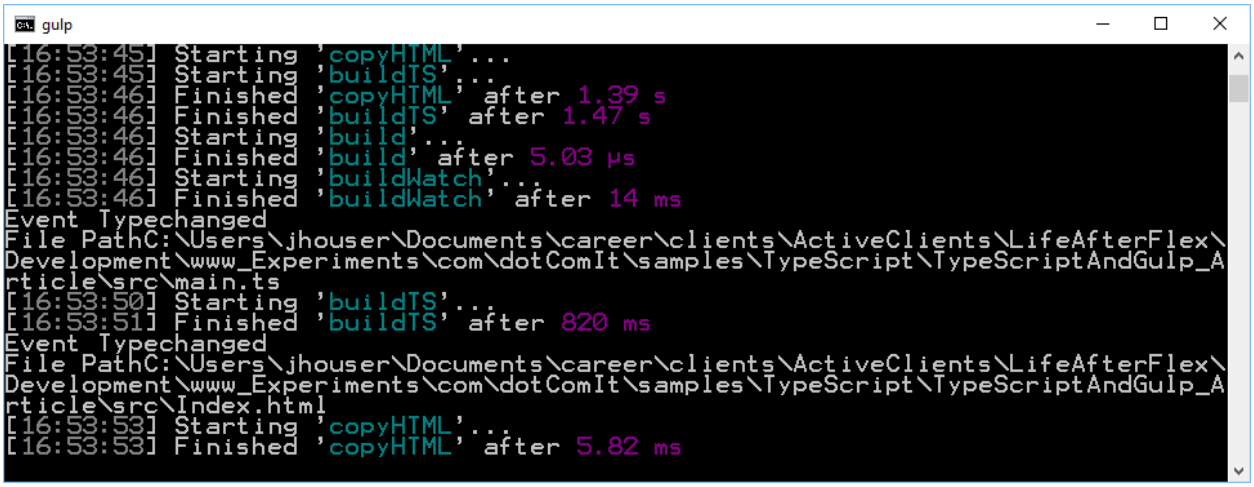
Run this script:

```
gulp buildWatch
```

You'll see something like this:



Notice that the console does not go back to a command prompt; the code is left running in the background. Change some code to see what happens:

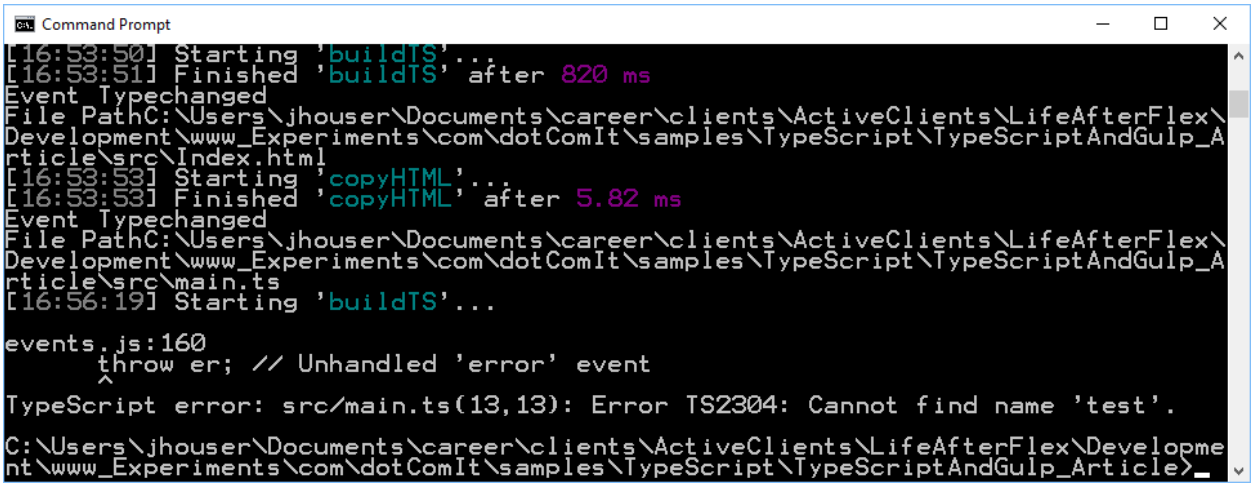


As you save files, the script automatically notices, and reruns the appropriate tasks. You'll get console information about the file that has changed.

My one complaint about this is that errors may cause the watch script to stop. If I type this in the main.ts file:

```
console.log(test)
```

And save it, I'll get this error:



The error is valid, and the test variable is not defined; but I wish the watch would not stop. I can correct the errors without having to restart my build script.

Create a Production Build

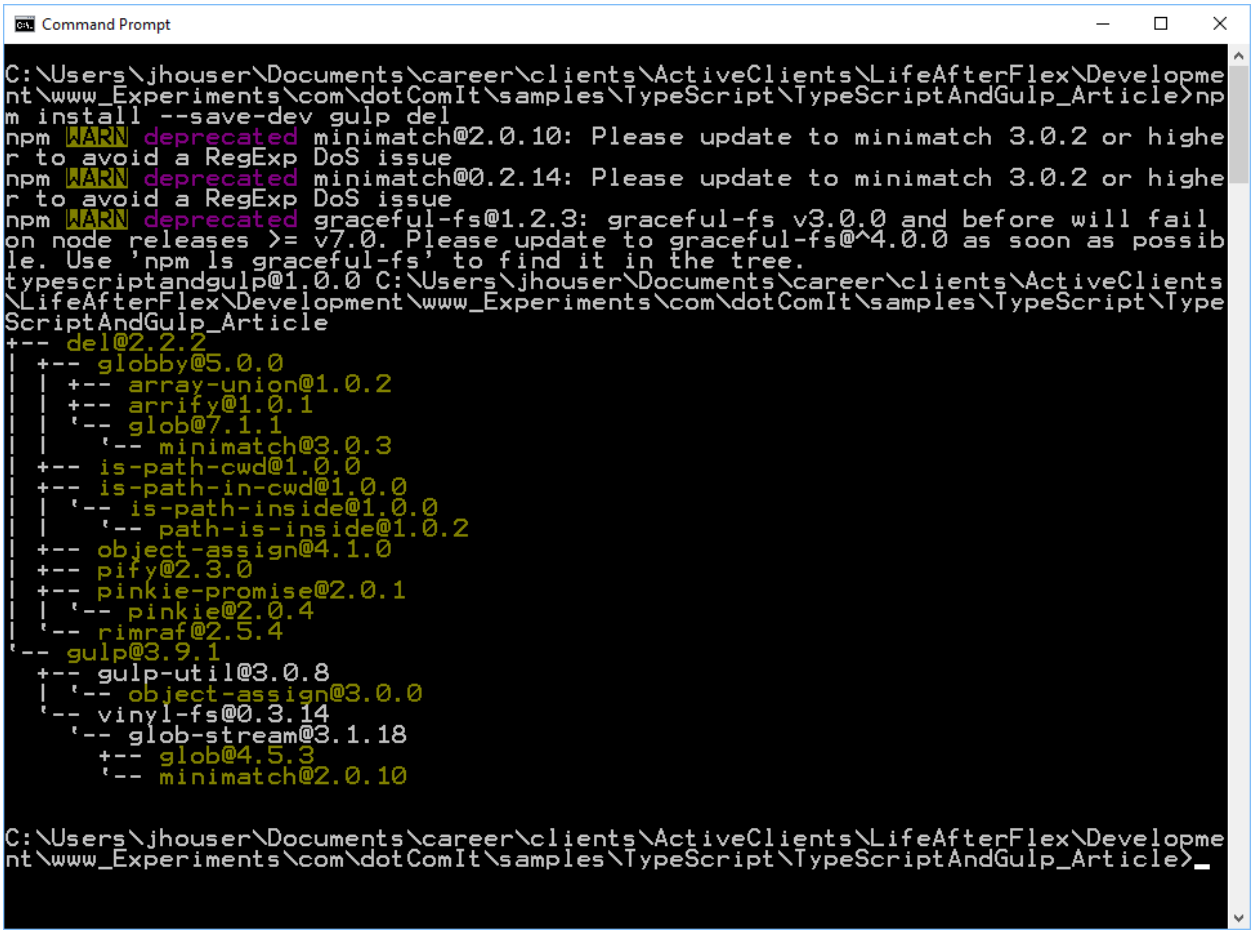
I want to create a Gulp task which will create a production build. This task will first need to delete all files in the build directory and then it will set the devMode variable to false and execute the build.

Create the Clean Task

The first step is to create a clean task. To do so, we'll use the del NodeJS plugin. Install it:

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

You should see an install window like this:



Now import it into the gulpfile.js:

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

Del is not a Gulp plugin, but can easily be used inside a Gulp task, similar to how Browserify is used inside a Gulp task, but we still use it.

First, create the deletePath:

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

This is a global variable. Next, create a clean task:

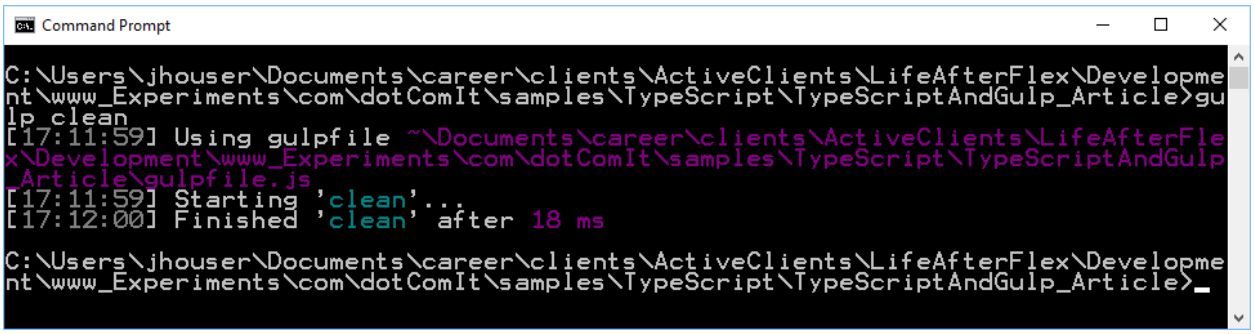
```
gulp.task('clean', function () {
    return del(deletePath);
});
```

This is a standard Gulp task setup, with the task named clean. Then the del() module is called with the deletePath variable passed in. That is all that is needed.

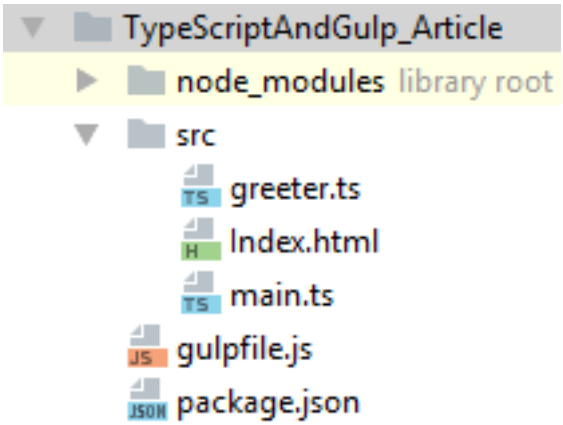
Run the task:

```
gulp clean
```

You'll see this as the result:



Easy enough! Take a look at your project files:



The build directory is completely gone. That is what we wanted.

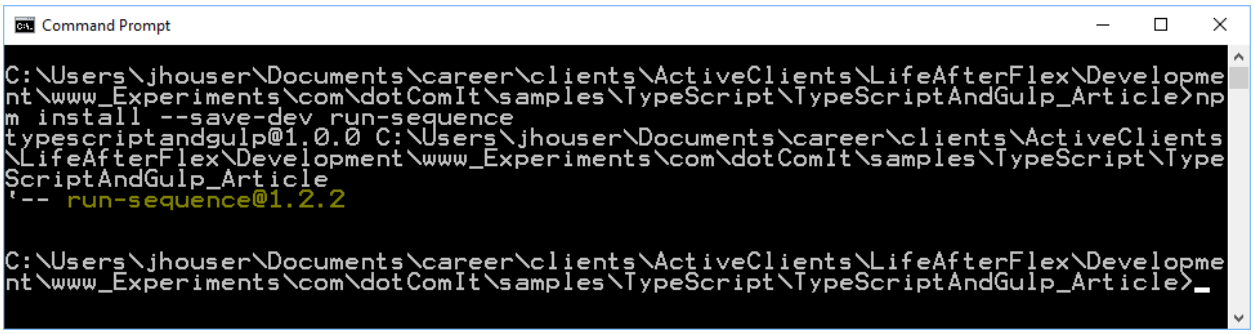
Create a Production Build Task

The last step in this white paper is to create a task for building a production build. For this, we'll need a Gulp plugin called run-sequence. The run-sequence plugin will let us easily run two Gulp tasks one after the other. Normally, Gulp Tasks would run in series. This would create an oddity if we start a build task before the clean task is complete.

Install run-sequence now:

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

You should see something like this in your console:



Now import the run-sequence plugin:

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

Now, create the buildProd task:

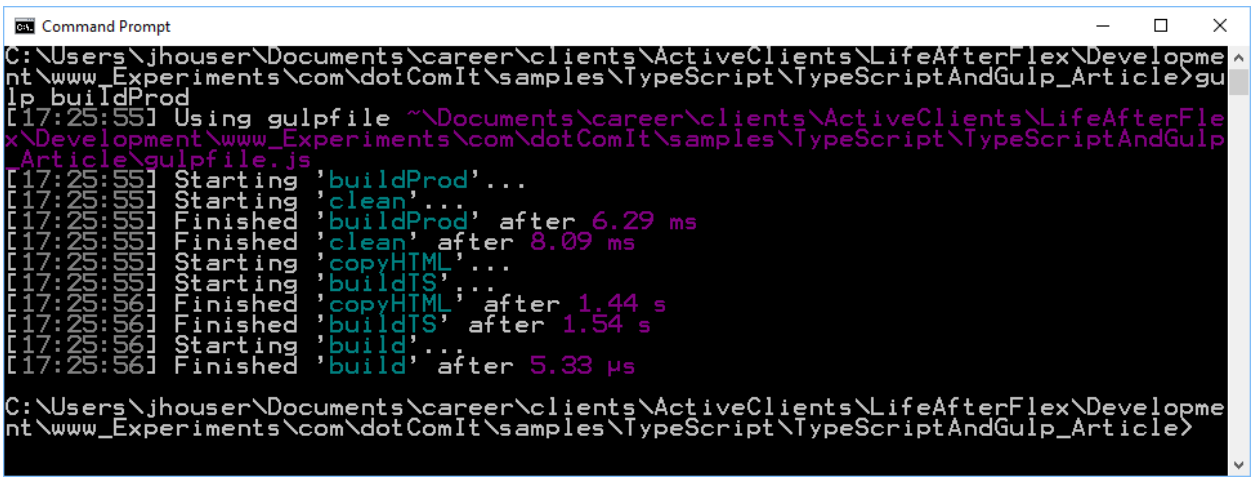
```
gulp.task('buildProd', function(){
    devMode = false;
    runSequence('clean', 'build');
});
```

The task sets the devMode variable to false. Then it runs the clean task. When the clean task is complete, the build task is run. Since devMode is set to false, source maps will not be created, but the UglifyJS plugin will run.

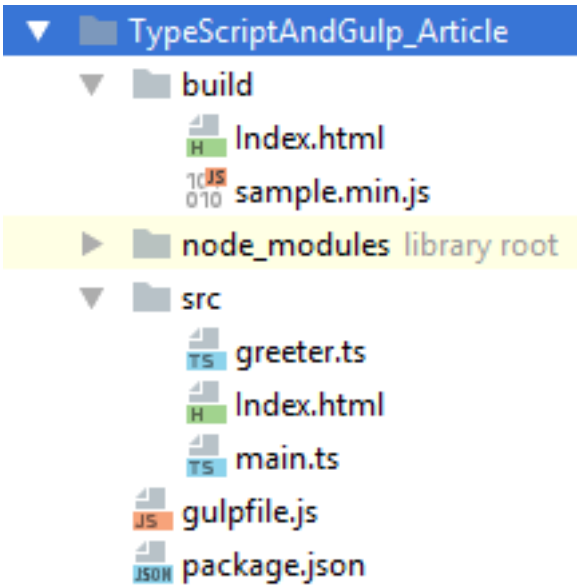
Try it out:

```
gulp buildProd
```

You should see this in the console:



Take a look at the directory structure:



The source maps were not created in the build directory, as expected. Open up the sample.min.js file:

```
!function r(e,n,t){function o(i,f){if(!n[i]){if(!e[i]){var
c="function"==typeof require&&require;if(!f&&c)return c(i,!0);if(u)return
u(i,!0);var l=new Error("Cannot find module '"+i+"'");throw
l.code="MODULE_NOT_FOUND",l}var
a=n[i]={exports:{}};e[i][0].call(a.exports,function(r){var
n=e[i][1][r];return o(n?n:r)},a,a.exports,r,e,n,t)}return
n[i].exports}for(var u="function"==typeof
require&&require,i=0;i<t.length;i++)o(t[i]);return
o}({1:[function(r,e,n){"use strict";function t(r){return"Hello from
"+r}n.sayHello=t},{}],2:[function(r,e,n){"use strict";function
t(r){document.body.innerHTML=o.sayHello(r)}var
o=r("./greeter");t("World")},{"./greeter":1}]},{},[2]);
```

The code is minimized and obfuscated as expected.

Final Thoughts

My intent in writing this white paper was to show how to compile TypeScript. We took on TypeScript because it was the first step in getting deeper into Angular 2, however there are other uses of TypeScript. Gulp has because one important part of the DotComIt build process for client applications, and I tried to detail a lot of options here.

This article should give you everything you need to setup your own environment for your own application development, but if you need help feel free to reach out.