Learn With

Knowledge to Boost Your Career

Angular 4 Collected Essays

Angular CLI

Custom Build Scripts

Debugging Techniques

Unit Testing

WWW.LEARN-WITH.COM

Part of the DotComIt Brain Trust

LEARN WITH ANGULAR 4 COLLECTED ESSAYS

By Jeffry Houser

https://www.learn-with.com

https://www.jeffryhouser.com

https://www.dot-com-it.com

About the Author

Jeffry Houser is a technical entrepreneur that likes to share cool stuff with other people.

In the days before business met the Internet, Jeffry obtained a Computer Science degree. He has solved a problem or two in his programming career. In 1999, Jeffry started DotComIt; a company specializing in custom application development.

During the Y2K era, Jeffry wrote three books for Osborne McGraw-Hill. He is a member of the Apache Flex Project, and created Flextras; a library of Open Source Flex Components. Jeffry has spoken all over the US. He has produced hundreds of podcasts, written over 30 articles, and written a slew of blog posts.

In 2014, Jeffry created Life After Flex; an AngularJS training course for Flex Developers. In 2016, Jeffry launched the Learn With series with books focusing on using AngularJS with other technologies. Jeffry has worked with multiple clients building Angular applications.

Contents

Preface

I was a Flex developer for a long time; however, Adobe's Flash Platform is no longer relevant. A smart developer will spend time educating himself on new technologies to keep up with a changing market, and to make sure he has healthy job prospects in the future. While cultivating these new skills, I decided to write about my experiences. With this series of books, you can leverage my experience to learn quickly.

This book is about my experiences building Angular applications. Angular is a JavaScript framework built by Google for building smart user interfaces. It is built on TypeScript and allows you to build dynamic views in HTML5. It is fully testable, which is important to many enterprise-level applications. It has a large developer community, ready to help with problems. I greatly enjoy building applications with Angular.

This book contains a series of essays around important parts of the HTML5 ecosystem that were not covered as part of the main series.

Introduction

What is this Book Series About?

The purpose of this series is to teach by example. The plan is to build an application using multiple technologies. These books will document the process, with each book focusing on a specific technology or framework. This entry is a companion to the Angular 4 books and covers HTML5 development topics that weren't used in the main books.

This book covers:

- **Creating Your Own Build scripts**: Build Scripts are the compilers of the HTML world. You used one throughout the main book. This chapter will show you how it was built.
- **Using the Angular CLI**: The Angular CLI has become the standard build script library for creating Angular applications. This chapter will show you how to use it yourself.
- **Application debugging**: Code isn't perfect the first time you write it; and this chapter will show you techniques to debug your HTML5 application.
- **Unit Testing**: One of the primary reasons for choosing Angular is that it creates easily testable code. This chapter will show you all the details.

These are important topics to add to your development toolbox.

Who Is This Book for?

Want to learn about building HTML5 applications? Are you interested in Angular or Bootstrap? Do you want to learn new technologies by following detailed examples with runnable code? If you answered yes to any of these questions, then this book is for you!

Here are some topics we'll touch upon in this book, and what you should know before continuing:

- **TypeScript**: This is the language behind Angular. It is a statically typed language that compiles to JavaScript. The more you know about it, the better. If you are not familiar with it yet, check out our tutorial lesson on learning the basics of TypeScript.
- **Angular**: If you read the main books of this series, then you should have more than enough Angular experience to understand it's usage here.
- **NodeJS**: We use these scripts to compile our TypeScript into JavaScript, process CSS, and copy files. Familiarity with NodeJS will be beneficial.
- **Jasmine**: This is a unit testing framework, and the book assumes you have no experience with it.
- **Karma**: This is a unit testing test runner. The book assumes you have no experience with it.

Common Conventions

I use some common conventions in the code behind this book.

- **Classes**: Class names are in proper case; the first character of the class in uppercase, and the start of each new compound word being in uppercase. An example of a class name is **MyClass**. When referencing class names in the book text, the file extension is usually referenced. For TypeScript files that contain classes the extension will be “ts”. For JavaScript files, the extension is “js”.
- **Variables**: Variable names are also in proper case, except the first letter of the first compound word; it is always lowercase. This includes class properties, private variables, and method arguments. A sample property name is **myProperty**.
- **Constants**: Constants are in all uppercase, with each word separated by an underscore. A sample constant may be **MY_CONSTANT**.
- **Method or Function Names**: Method names use the same convention as property names. When methods are referenced in text, open and close parentheses are typed after the name. A sample method name may be **myMethodName()**.
- **Package or Folder Names**: The package names—or folders—are named using proper case again. In this text, package names are always referenced as if they were a directory relative to the application root. A sample package name may be **com/dotComIt/learnwith/myPackage**.

Caveats

The goal of this book is to help you become productive creating HTML5 apps with a focus on Angular. It leverages my experience building business apps, but is not intended to cover everything you need to know about building HTML5 Applications. This book purposely focuses on the tool chain. If you want to know more about Angular explicitly, check out original series. You should approach this book as part of your learning process and not as the last thing you'll ever need to know. Be sure that you keep educating yourself. I know I will.

Want More?

You should check out this book's web site at www.learn-with.com for more information, such as:

- **Source Code**: You can find links to all the source code for this book and others.
- **Errata**: If we make mistakes, we plan on fixing them. You can always get the most up-to-date content available from the website. If you find mistakes, please let us know.
- **Test the Apps**: The web site will have runnable versions of the app for you to test.
- **Bonus Content**: You can find more articles and books expanding on the content of this book.

Chapter 1: Dissecting Build Scripts

The Angular CLI project has become the most common approach for building Angular 4 applications. When I started writing the Learn With series on Angular 4, it had not become the standard. As such I created my own workflow process. I based it around Gulp, which I used extensively for building AngularJS 1.x applications. I thought it would be a great learning exercise to explain more details. This article will dissect the scripts I wrote and explain how to use them.

All the build scripts are in the Learn With application repository, however if you want the sample application too, get it from this GitHub repository.

Build the Application

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

Code the TypeScript

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 applications, I use a directory structure that mimics the package structure I used to use when building applications in other technologies. Classes are put in a directory structure like **'com/dotComIt/*appName/somethingDescriptive*'**. The '*appName*' will be a descriptive name for the application. 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(AppModul
```

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** in the same directory:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-
browser';
import { HashLocationStrategy, LocationStrategy }
from '@angular/common';
import { AppComponent }  from './app.component';
import { AppRoutingModule } from
'../nav/routing.module';


@NgModule({
  imports:      [ BrowserModule, AppRoutingModule
],
  declarations: [ AppComponent ],
  providers:    [{provide: LocationStrategy,
useClass:HashLocationStrategy}],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

This file imports four Angular classes from three Angular libraries. The core library and platform-browser library are used for loading and running the app. The **HashLocationStrategy** and **LocationStrategy** classes are used for internal navigation. We have two custom libraries: **AppComponent** and **AppRoutingModule**. The **appComponent** is the primary component for our application, and the **AppRoutingModule** will handle navigation duties. Then the **@ngModule** annotation is created. This is metadata that defines the application. It imports the **browserModule** and **AppRoutingModule**. The **imports** property is used to load other modules into this one so it's functionality, such as components or

providers, can be available. The **declarations** property is used to load components as part of this module, and the **AppComponent** is loaded here.

The **providers** property is used to load the hashing strategy. This controls how the routing module will change the URL as you move to different screens across the app. The **bootstrap** property is used to load the application's main component, **AppComponent**. Finally, 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>
             <router-outlet></router-outlet>`,
})
export class AppComponent  { name = 'World'; }
```

This file imports the **Component** class from the **@angular/core** library. It creates the **@Component** annotation; which defines the modules main component. The selector refers to the HTML tag that will be used to access this component. The template defines an in-line template that will replace the tag after the application is bootstrapped. The template starts with a simple hello world app, but also includes a tag called **router-outlet**, which is a special component as part of the Angular routing module. Finally, the **AppComponent** class is exported. This class defines the **name** variable as 'World', which is accessed in the view to create the hello world application.

That completes the TypeScript portion of the application to load the Angular libraries and an initial component. We'll still need to create the **routing.module.ts** and a couple of routes.

Create Angular Routes

The sample routes will consist of two Angular components, and a

routine module. First, create the routing module file, named **routing.module.ts** in the **com/dotComIt/learnWith/nav** directory. This app will create an independent module for the purposes of routing. First import some libraries:

```
import {NgModule} from '@angular/core';
import {RouterModule, Routes} from
'@angular/router';
import {FirstComponent} from
"../views/first/first.component";
import {SecondComponent} from
"../views/second/second.component";
```

First, the **NgModule** is imported from **@angular/core**. Then the **RouterModule** and **Routes** classes are imported from **@angular/router**. Finally, two custom components, **FirstComponent** and **SecondComponent** are imported. We'll create those shortly.

Create a constant to define the routing details:

```
const routes : Routes = [
    { path: 'first', component: FirstComponent },
    { path: 'second', component: SecondComponent
}
    { path: '', redirectTo: 'first', pathMatch:
'full' },
];
```

The routes constant is of value **Routes**, which is an array of **Route** objects. Each route is a collection of properties that define how that route will act. The route "first" will load the **FirstComponent** and display it where the **router-outlet** is. The router "second" will load **SecondComponent** and display it where the **router-outlet** is. The final entry will redirect an empty path to the first path.

Next, create the **@NgModule** annotation:

```
@NgModule({
    imports: [ RouterModule.forRoot(routes) ],
    declarations: [ FirstComponent,
```

```
SecondComponent ],
    exports: [ RouterModule ]
})
```

This sets up the **RouterModule** using the **forRoot()** method and passing in the **routes** constant. It loads the **FirstComponent** and **SecondComponent** using the declaration, and exports the **RouterModule** for use by other modules.

Finally, export the main class:

```
export class AppRoutingModule {}
```

This is the class that was loaded, and imported, into the main application module.

Now, create the **FirstComponent** in the **com/dotComIt/sample/views/first** directory. The component will be comprised of three files: A CSS file, an HTML Template file, and a TypeScript file. Create the CSS file first, named **first.component.css**. This is just a placeholder for future user, but let's add something in it for proof of principle:

```
.no-padding{
    padding:0;
}
```

Now create **first.component.html**:

```
<h1>First View</h1>
<a [routerLink]="['/second']">Go to Second
View</a>
```

This HTML template includes a header, and a link to the second view. The **routerLink** property is used, which is Angular magic behind the scenes for automatically loading a new route when the anchor is clicked.

Finally, create **first.component.ts**:

```
import { Component } from '@angular/core';
@Component({
    selector: 'first',
```

```
    templateUrl :
'./com/dotComIt/sample/views/first/first.componen
    styleUrls: [
'./com/dotComIt/sample/views/first/first.componen
]
})
export class FirstComponent { }
```

It imports the **Component** class from **@angular/core**. The **@Component** annotation is used to define the component. We specify the **selector**, a **templateUrl**, and **styleUrls**. The **tempalteUrl** is a location from the main index file—or application root—to the template. The **styleUrls** contains an array of paths to style sheets used in this component. Normally in HTML applications, style sheets are shared across components, but I've found that Angular uses some magic to limit the style sheets specified in the **styleUrls** from affecting other components that do not load it. It is a nice easy way to avoid conflicts with similarly named styles.

Create the second component in the **com/dotComIt/sample/views/second** directory. The file will consist of three similar files. First, **second.component.css**:

```
.genericCenteredText{
    text-align:center;
}
```

This is just some place holder style to test our processing code. Then the template file, **second.component.html**:

```
<h1>Second View</h1>
<a [routerLink]="['/first']">Go to First View</a>
```

This displays a header for the second view, and a link back to the first view.

Finally, the **second.component.ts** file:

```
import { Component } from '@angular/core';
@Component({
    selector: 'second',
    templateUrl :
```

```
'./com/dotComIt/sample/views/second/second.compon
    styleUrls: [
'./com/dotComIt/sample/views/second/second.compon
]
})
export class SecondComponent { }
```

When the app is compiled and then loaded in a browser, we will be able to navigate between the first and second view using the links.

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 property:

```
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'
},
```

The **map** configuration tells the code “when the code imports the ‘property’, go look in the ‘value’ to find the directory for the library”. The main angular libraries are listed. Explanation of the Angular libraries is beyond the full 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 the **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.

Write 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>
```

First, add a **base href** tag to the document head:

```
<script>document.write('<base href="' +
document.location + '" />');</script>
```

The **base** tag is required by the Angular router and defines the current page which will be used as a point of reference when the router changes the URL. This uses some JavaScript magic to determine the current location of the page.

Import our application's main style sheets:

```
<link rel="stylesheet" href="app.min.css">
```

Later we'll use some process to compile all the non-Angular-Component specific style sheets into one small one.

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 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 load. The rest of this article will address those omissions.

Install Angular Libraries

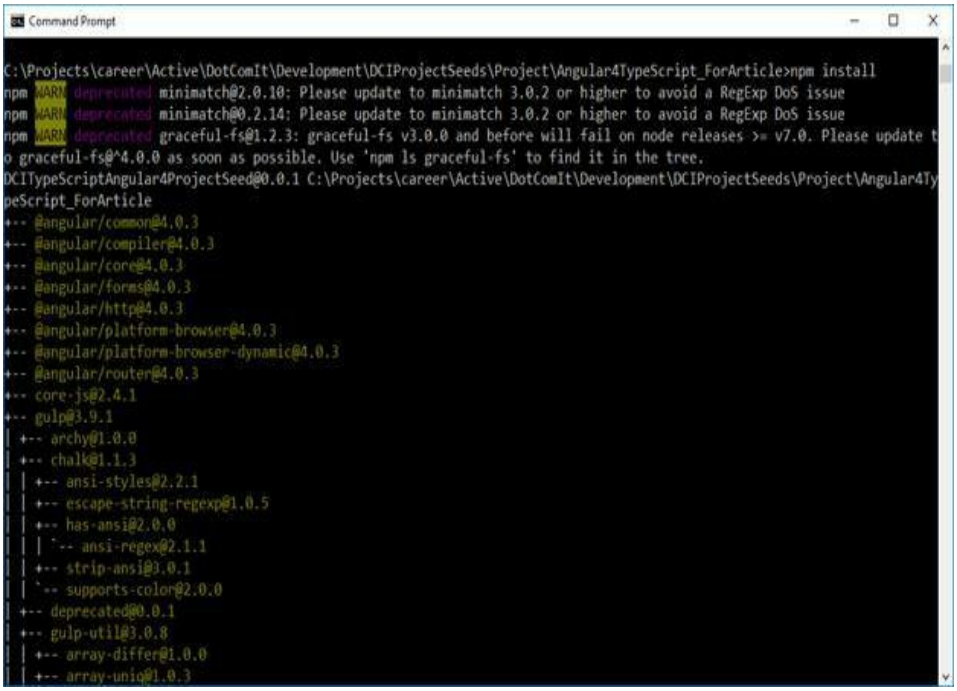
Gulp runs on top of NodeJS. If you don't already have it installed, do it now. The formal instructions or getting setup will be more helpful than anything I'd tell you in this article. After that, we need to create a **package.json** in the root directory. Copy and paste this one:

```
{
  "name": "TypeScriptAngular4Sample",
  "version": "0.0.1",
  "description": "TypeScript Angular 4 Sample for
Article.",
  "author": "Jeffry Houser",
  "license": "ISC",
  "dependencies": {
    "@angular/common": "~4.0.1",
    "@angular/compiler": "~4.0.1",
    "@angular/core": "~4.0.1",
    "@angular/forms": "~4.0.1",
    "@angular/http": "~4.0.1",
    "@angular/platform-browser": "~4.0.1",
    "@angular/platform-browser-dynamic":
"~4.0.1",
    "@angular/router": "~4.0.1",
    "systemjs": "0.19.40",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.8",
    "rxjs": "5.0.1",
    "zone.js": "^0.8.4"
  },
  "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 additional Gulp plugins, we'll install them separately.

Compile 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

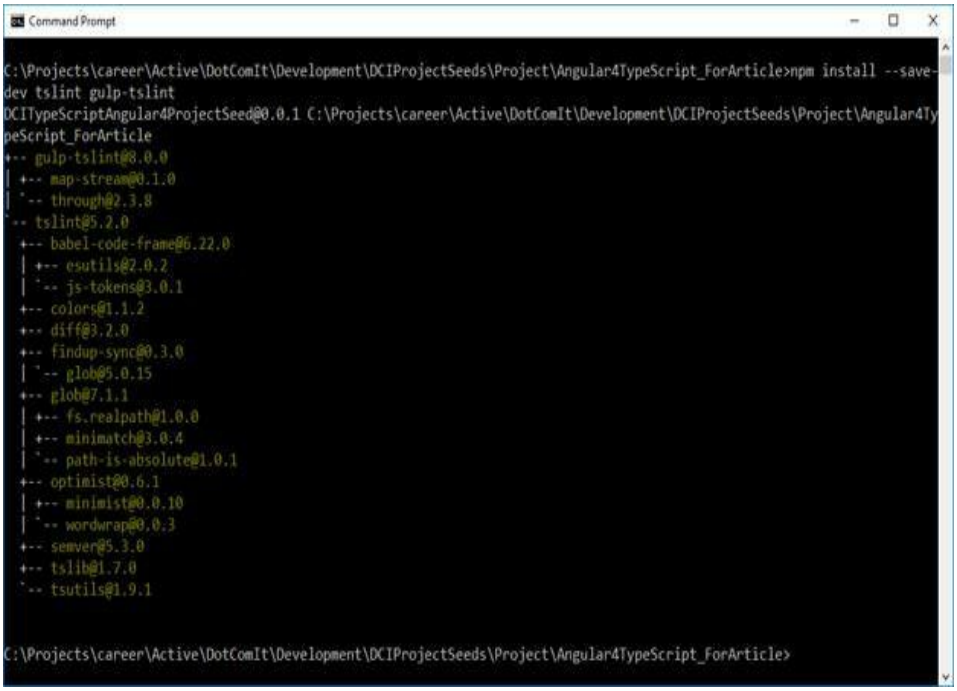
We'll need to install a few new NodeJS Modules for this section:

- **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
```

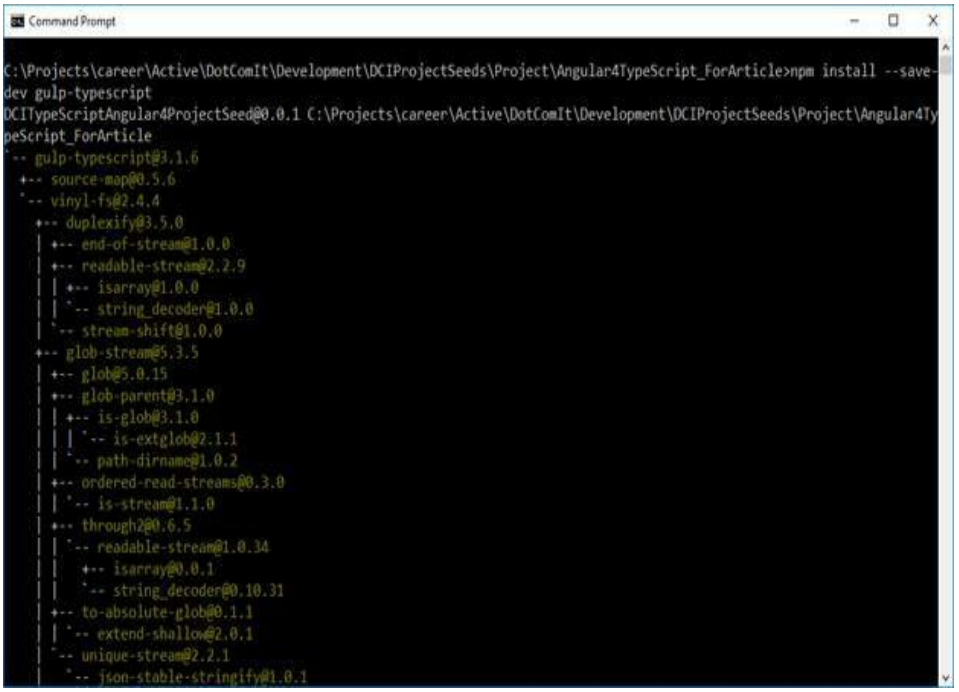
You should see results something like this:



Then install gulp-typescript:

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

You'll see results like this:



That installs the dependencies, now it is time to setup the Gulp file.

Create a Gulp Configuration Module

I want to encapsulate all the configuration options, or modifiable variables, into a separate Node component than the main Gulp file. Create a file named **config.js** in the project root. This will be split up into three separate sections, each one representing an object of relevant values.

First, create a **baseDirs** object:

```
var baseDirs = {
    sourceRoot : "src/",
    codeRoot : 'com',
    destinationPath : 'build',
};
```

This contains the basic directories structure for finding the source code root ‘src’, the Angular application root ‘com’, and the destination path ‘build’. This object will be reused inside this config module, and as part of our main Gulp script. Now, create a config object:

```
var configObject = {
};
```

This will contain various values that we want to configure, but for now leave it empty. We’ll populate it throughout this article.

Finally, create a **staticConfig** value:

```
var staticConfig = {
};
```

This will contain values, such as the location of angular modules, that you probably won’t want to change, but I encapsulated them just in case. Now, export the config object:

```
exports.config =
Object.assign(baseDirs,configObject,staticConfig)
```

The **Object.assign()** combines all three objects into a new one which is exported to be used as a module inside a different NodeJS script.

The idea of having a config file separate from the scripts file is so we can easily make changes to directories without affecting the scripts.

Lint 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 config object:

```
var config = require("./config").config;
```

This will give us access to all the config values.

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.** Put this in the **config.js** module as part of the **configObject**:

```
typeScriptSource : [baseDirs.sourceRoot +
"**/*.ts"]
```

Notice, this value references the **baseDirs** object's **sourceRoot** value. The value is a wildcard glob that will cover all existing TypeScript files in the main source 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());
```

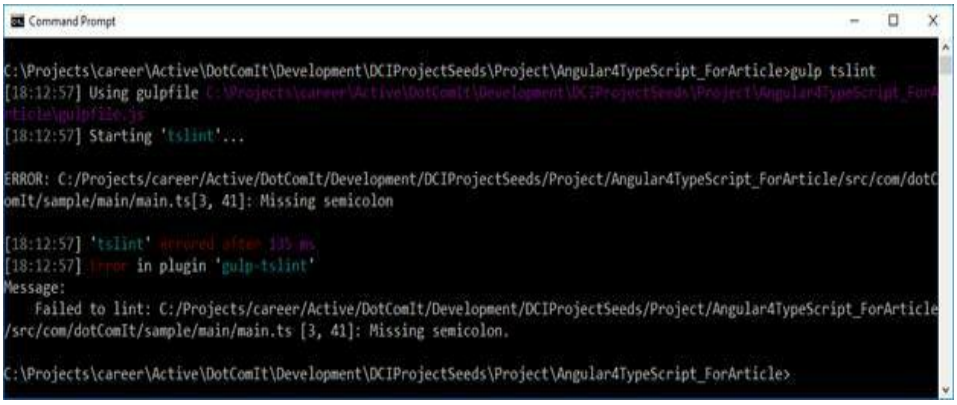
That is the completed task. Before we run it, we'll need to configure it. Specific configuration options are beyond the scope of this article. Put this tslint.json file, in your root directory and you'll be fine. The file comes from the official Angular Quickstart documentation.

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, 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 4. The second important thing here is the target, es5. This stands for ECMAScript 5; which is a JavaScript implementation that most browsers support.

The second important item in the **tsconfig.json** file is:

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

The exclude option tells the type script project not to process files in the **node_modules** directory. 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 create the type script process:

```
var tsResult = gulp.src(config.typeScriptSource)
                   .pipe(tsProject());
```

It uses the same **typeScriptSource** variable that the **tslint** task used. Then, it adds the **tsProject** as a Gulp pipe. Then it takes the **js** results and copies them to their destinations:

```
return tsResult.js

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

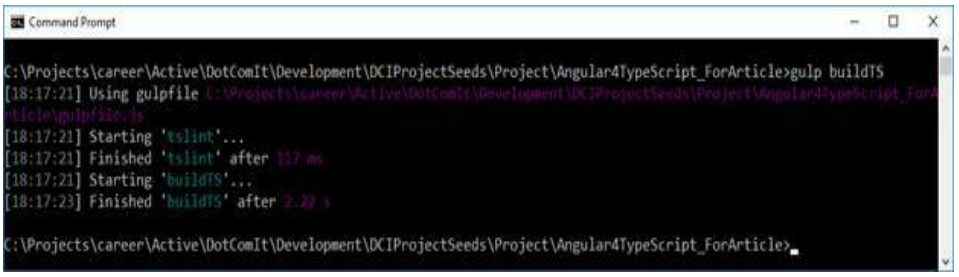
What the full script does is take all the TypeScript files in our source

directory, run them through the gulp-typescript compiler, and save the output to a **build** directory.

Run the script:

```
gulp buildTS
```

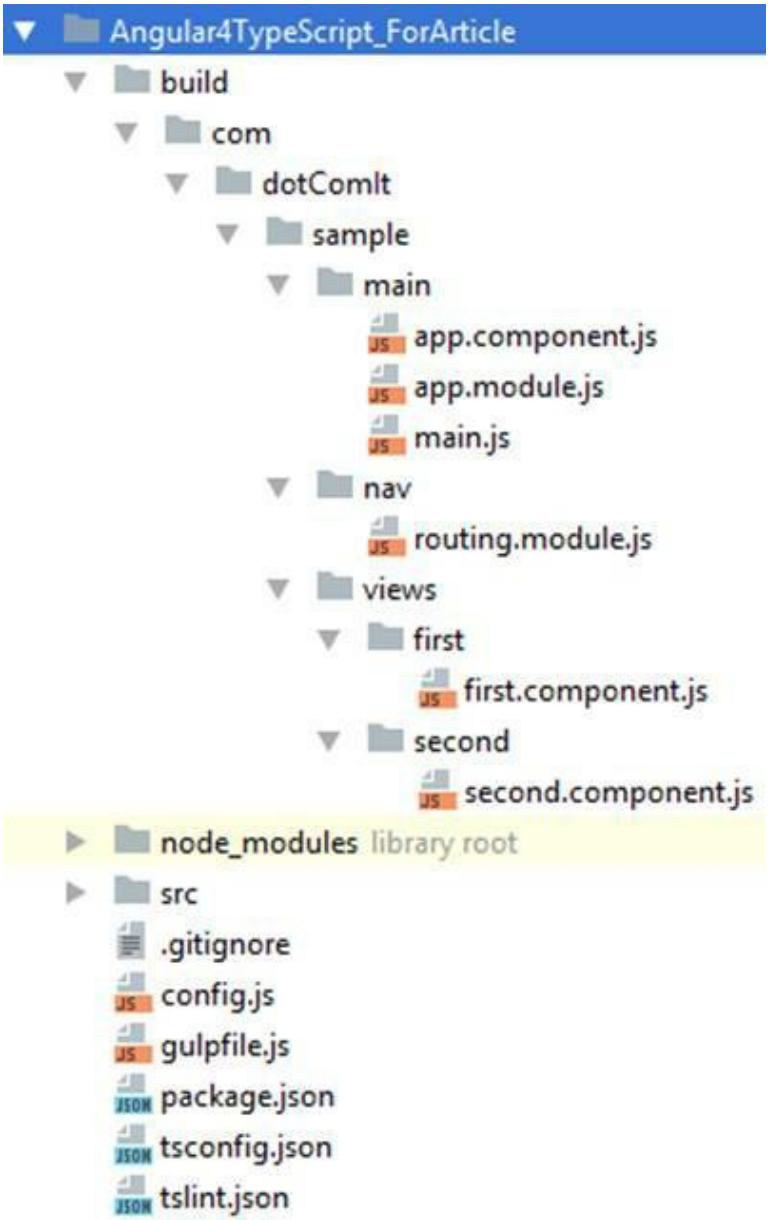
You'll see results similar to this:



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";
Object.defineProperty(exports, "__esModule", {
value: true });
var platform_browser_dynamic_1 =
```

```
require("@angular/platform-browser-dynamic");
var app_module_1 = require("./app.module");
platform_browser_dynamic_1.platformBrowserDynamic
```

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 we told TypeScript to use.

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 Static Assets

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 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. Add this to the **staticConfig** object in the **config.js** file:

```
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 4 applications such as rxjs, 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. The previous paths are all relative to the **node_modules** directory. Add a glob to point to the **node_modules** directory:

```
nodeModulesSource : "node_modules/**"
```

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

```
destinationPathForJSLibraries =
baseDirs.destinationPath + '/js';
```

This goes in the **configObject** part of the **config.js** file. It makes it easy for you to modify things for your own preferred directory structure. I put all the external libraries in a **js** folder as an organizational tactic.

Back to the **gulpfile.js**, create the Gulp task to copy the Angular libraries:

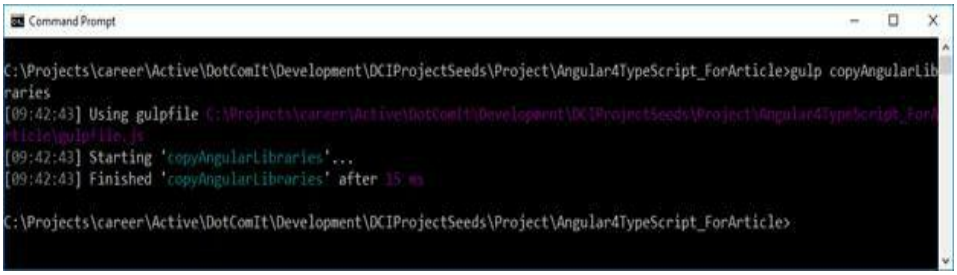
```
gulp.task('copyAngularLibraries', function () {
    gulp.src(config.angularLibraries, {cwd:
config.nodeModulesSource})

.pipe(gulp.dest(config.destinationPathForJSLibrar
});
```

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.

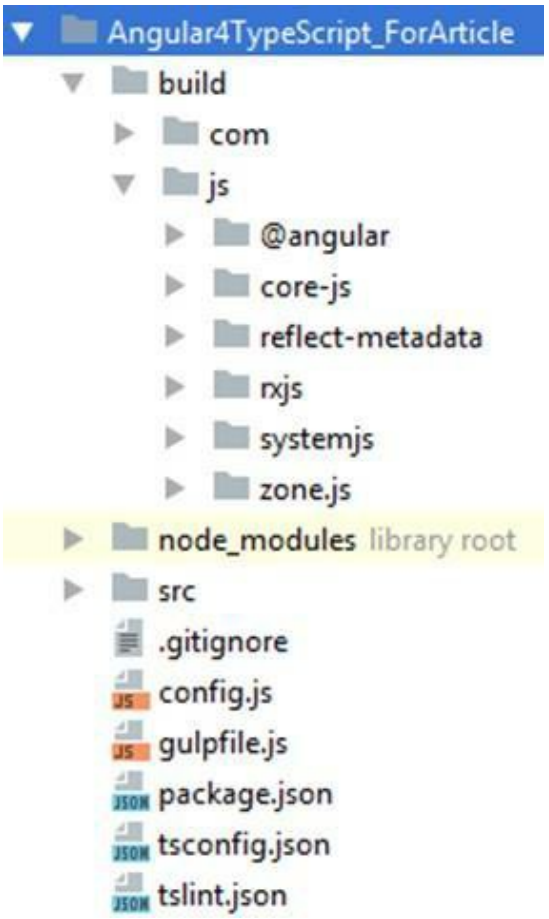
Run this task:

```
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. In the **config.js file**, create a glob array to find the JS files as part of the **configObject**:

```
javaScriptLibraries : [baseDirs.sourceRoot +
'js/**/*.js'];
```

Then create the task:

```
gulp.task('copyJSLibraries', function () {
```

```
    gulp.src(config.javaScriptLibraries)

.pipe(gulp.dest(config.destinationPathForJSLibrar
});
```

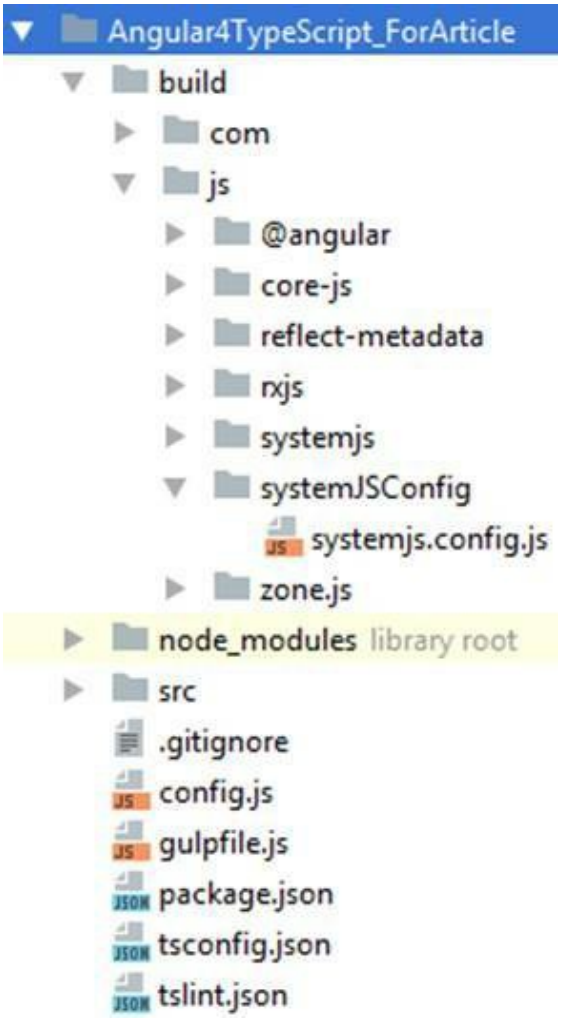
This is a simple task that specifies the input source and the destination. It uses the same destination variable that was used for the Angular libraries in the previous section.

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/js** directory.

Copy HTML Files

The app has a few HTML files. The primary one is the **index.html**. But, we also have two HTML template, one for **FirstComponent** and one for **SecondComponent**. We need a task copy all HTML Files from the **src** directory to the **build** directory. In the **configObject** of the **config.js** file, create an **HTMLSource** variable:

```
htmlSource : [baseDirs.sourceRoot + '**/*.html'],
```

It is a Glob that finds all the HTML files in the **sourceRoot**.

Then create the task:

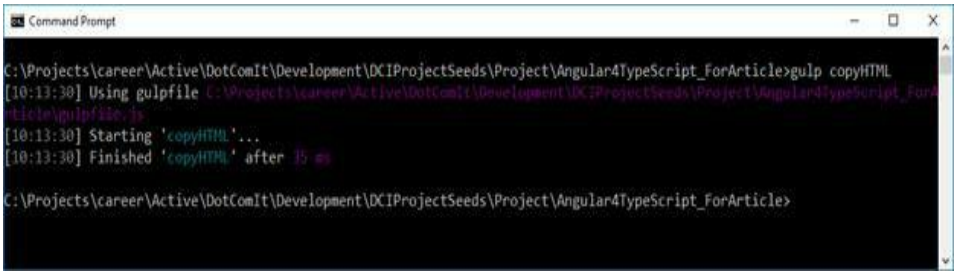
```
gulp.task('copyHTML', function () {
    return gulp.src(config.htmlSource)
        .pipe(gulp.dest(config.destinationPath));
});
```

The task is named **copyHTML**. It is a simple task that specifies the input source and the destination. It uses the same destination path 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.

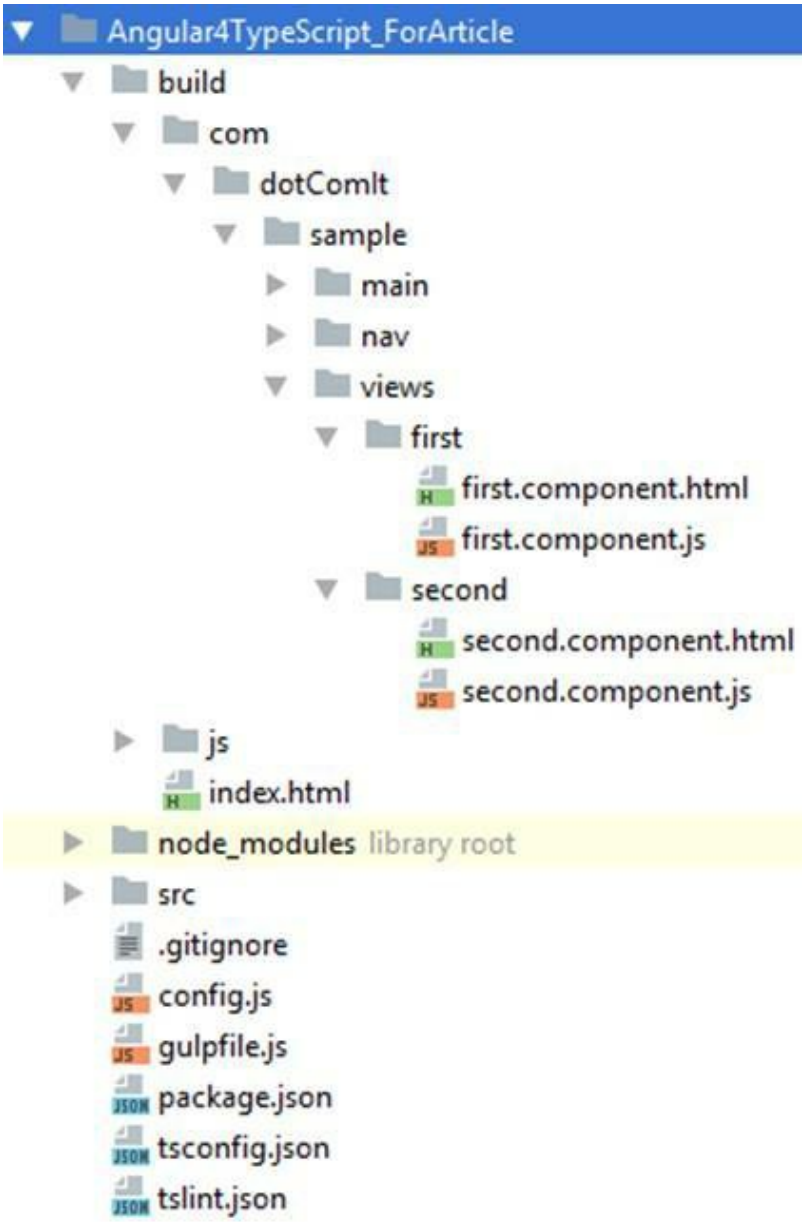
Run the task:

```
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. You can also see the HTML templates in the **build** directory. Their directory structure was preserved.

Process CSS Files

There are two types of CSS files we need to process. First, we want global application specific CSS files. We want to run those through a processor and combine them into a single file. The second is the Angular template CSS files. We want to process and minimize them, but we do not want to combine them into one—we just want to copy them from the **src** directory to the **build** directory while maintaining their directory structure. Angular will load the CSS files on demand and apply the styles to the component templates.

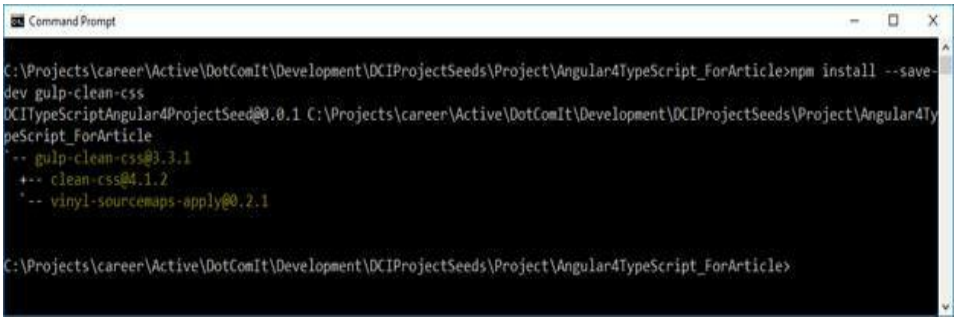
Install Plugins

We are going to install two separate Gulp file plugins to process CSS. The first will be a CSS processor named clean-css. The second will be a file concatenator, gulp-concat.

First, install clean-css:

```
npm install --save-dev gulp-clean-css
```

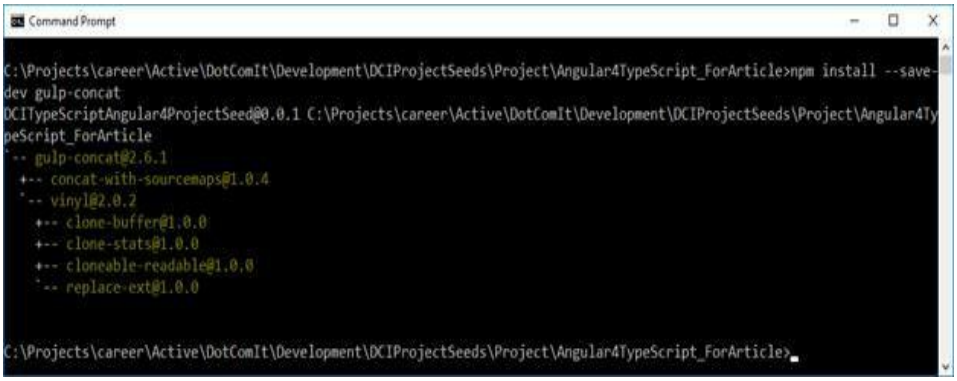
You'll see something like this:



Now install gulp-contact:

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

You'll see something like this:



That covers the prerequisites. Load them into the **gulpfile.js**:

```
var cleanCSS = require("gulp-clean-css");
var concat = require('gulp-concat');
```

Now we can use these plugins as part of our Gulp scripts.

Process Global CSS

This section will create a task to process all the CSS that is not part of the com directory. The assumption is that everything in the com directory will relate to Angular components loaded with a **styleUrls** property. Everything else will be part of the main application.

We'll need to create a few config values in the **config.js**. Create a source Globs for the CSS:

```
cssSource : [baseDirs.sourceRoot + '**/*.css',
             '!' + baseDirs.sourceRoot +
baseDirs.codeRoot + '/**/*.css'],
```

This uses the **baseDirs.sourceRoot** to refer to the source directory and find all the CSS files contained therein. But there is a second piece to this glob, which points to the **baseDirs.codeRoot**. This glob has an exclamation point in the front, telling the Glob to ignore files in this directory. So, this Glob says to find all the CSS files in the **src** that are not in the **com** application directory.

We also need to specify a destination file:

```
cssDestinationFile : 'app.min.css',
```

You may remember earlier in this article where we loaded this CSS

file in the **index.html** file. We are finally creating it.

Back to the **gulpfile.js**, create a **processCSS** script:

```
gulp.task('processCSS', function () {
    gulp.src(config.cssSource, { base: '.' })
        .pipe(cleanCSS())
        .pipe(concat(config.cssDestinationFile))
        .pipe(gulp.dest(config.destinationPath))
});
```

This script finds the **cssSource** as described above. It runs the **cleanCSS()** processor on it. This will process the files to minimize them by removing white space and will perform other optimizations. Then the **concat** library is called to combine all the CSS files into one. Finally, the destination is set using the **destinationPath** config value. This will put the new CSS file in the root of the build directory.

Technically, we haven't created any CSS outside of the main components, so this doesn't do much. However, let's create a few sample files for proof of principle. In the **src** root add a file named **styles.css**:

```
* {
    font-family: Arial, Helvetica, sans-serif;
}
body {
    margin: 2em;
}
```

This sets up some basic margins and font usage.

To test the concatenation portion of this, create a **styles** directory in the **src**. Add a file named **other.css** add this style in it:

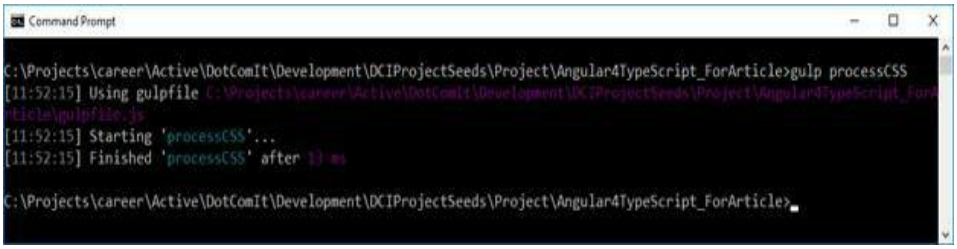
```
button {
    color: #888;
    font-family: Cambria, Georgia;
}
```

It just styles buttons with a color and specialized fonts.

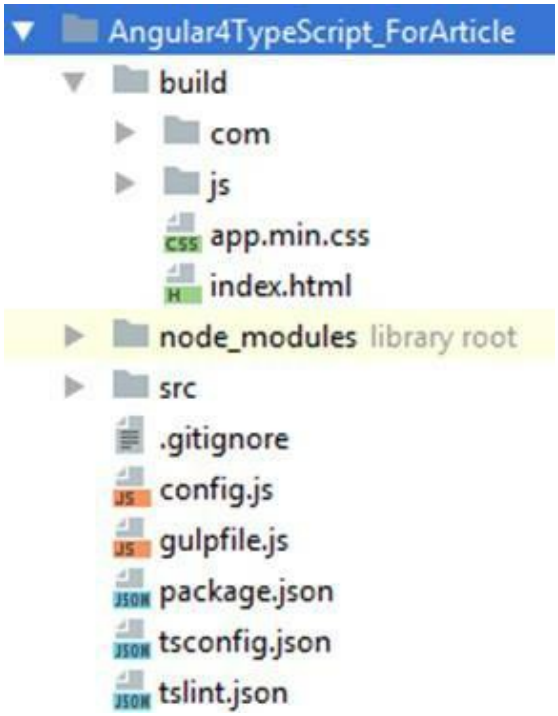
You can run this:

```
gulp processCSS
```

And you'll something like this:



Look at the directory structure and you'll see the **app.min.css**:



Open the file:

```
*{font-family:Arial,Helvetica,sans-
serif}body{margin:2em}
button{color:#888;font-family:Cambria,Georgia}
```

With such simple CSS, we just see some line breaks and white space

removed. More advance CSS would have other optimizations. I've seen borders and padding values re-written to use short hand instead of long-hand definitions. I've seen colors changed from a color name to a hex value, or vice versa depending upon the length. Sometimes I've even noticed styles rewritten so that similar styles use inheritance for shared elements.

Process Angular CSS Files

We want to process the Angular component files a bit differently. We do want to process them with cleanCSS, but we don't want to concatenate them into one big file. When they move from the source to the build directory we want to retain their directory structure so that Angular can properly load them based on the **styleUrls** property.

Open up the **config.js** file and edit the **configObject**:

```
cssStyleURLsSource : [baseDirs.sourceRoot +
baseDirs.codeRoot + '/**/*.css'],
destinationPathForCSSStyleURLs :
baseDirs.destinationPath + '/' +
baseDirs.codeRoot
```

The first value, **cssStyleURLsSource** is a Glob that will find all the CSS files in the **src/com** directory. The **destinationPathForCSSStyleUrls** specifies the final location for the files which is in **build/com**. The Gulp script will know to retain the directory structure.

Here is the Gulp script to **copyAngularCSS**:

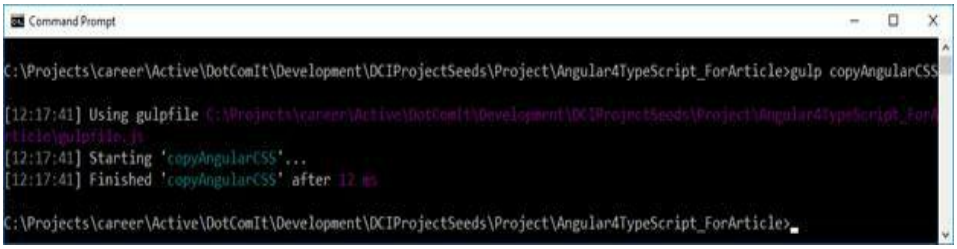
```
gulp.task('copyAngularCSS', function () {
    gulp.src(config.cssStyleURLsSource)
        .pipe(cleanCSS())

.pipe(gulp.dest(config.destinationPathForCSSStyle
});
```

The source is specified with the **cssStyleURLsSource** value. Then the **cleanCSS()** function is run to process the files. Finally, the files are copied to the destination.

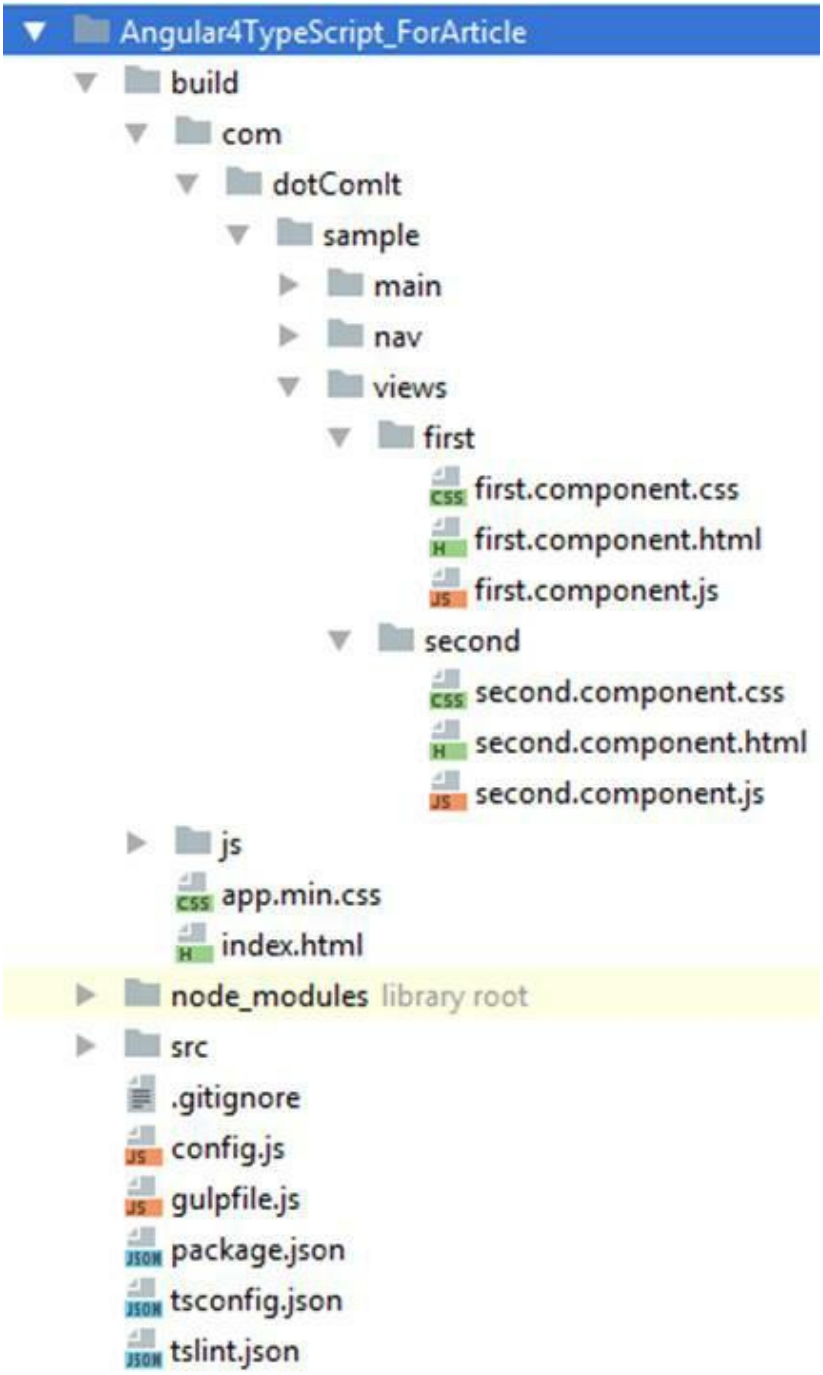
Run the script at your command line:

```
gulp copyAngularCSS
```

You'll see something like this:



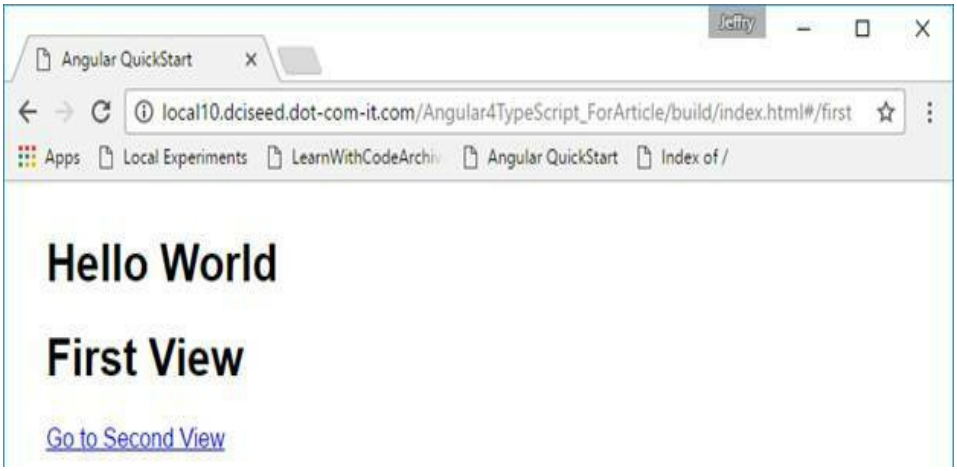
Look at the directory structure to see the CSS files:



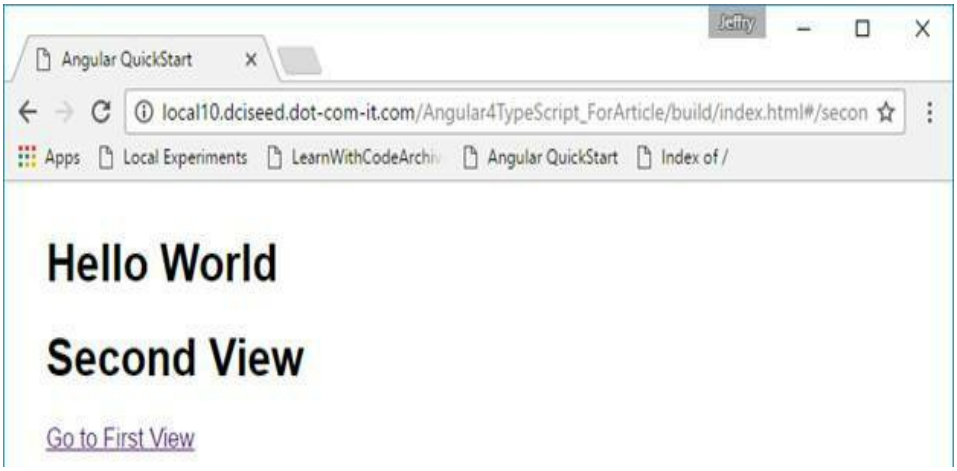
Run the App

With all the TypeScript compiled, HTML files copied, and CSS

processed we can run the app now:



The app loads, the router library changes the browser URL and the state is load. Click the 'Go to Second View' link:



The second view loads, the URL is changed. Not bad, but there is still a lot we can do to tweak our settings.

Run Both CSS Processes in One Task

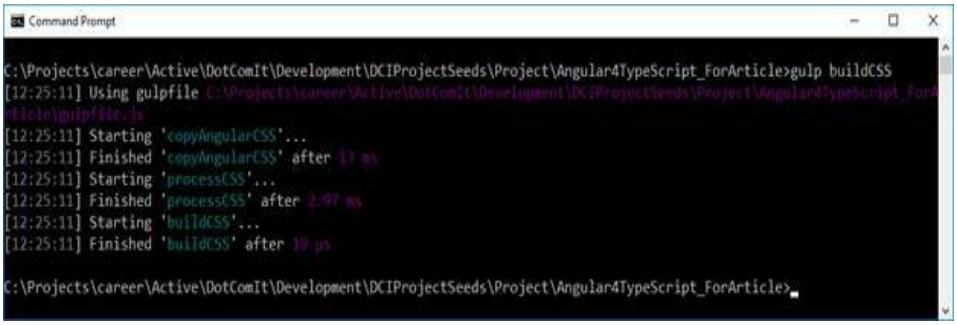
We built two Gulp tasks to process CSS in two different ways. But, sometimes we may want to run the two tasks in one command. Create a new task:

```
gulp.task("buildCSS", ['copyAngularCSS',
'processCSS']);
```

This task is named **buildCSS**. It doesn't have any task functionality of its' own, it just processes the previous two tasks we created. Run it:

```
gulp buildCSS
```

You'll see results like this:



This is just a helper task, but many times I find helper tasks to be beneficial.

Handle Source Maps

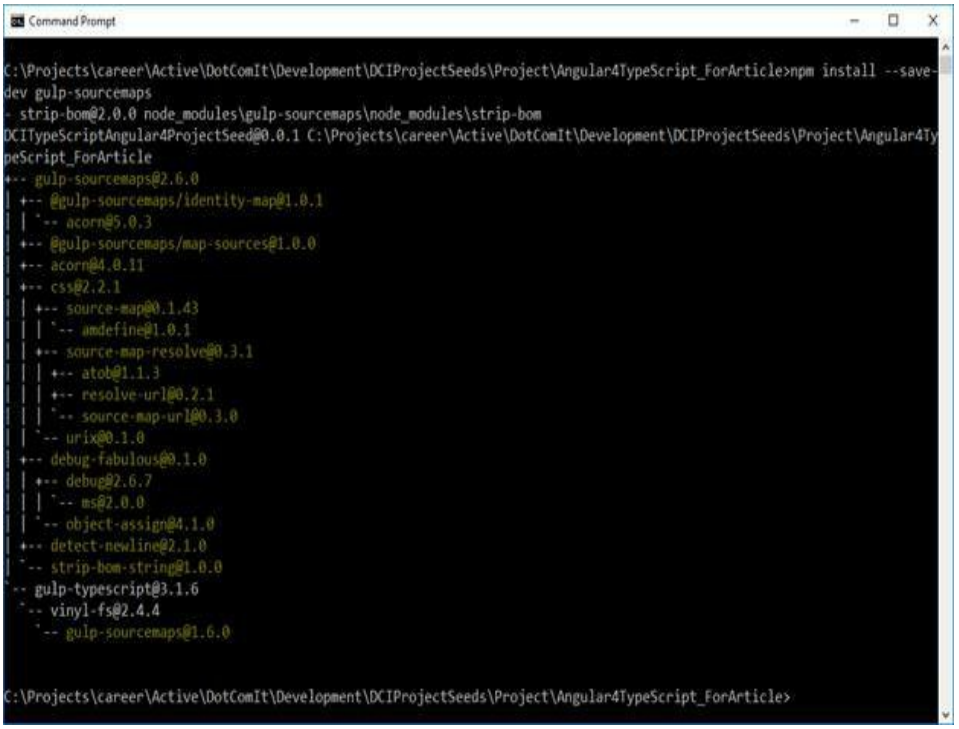
Since the TypeScript goes through a compilation process, the code running in the browser is not the same as the code we wrote. It has been translated from one language, TypeScript, to another, JavaScript. When an error occurs how can we debug it? The answer is to create a source map. We can look at the 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. We can even use Source Maps on our processed CSS files.

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 TypeScript 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. Put it in the **config.js** file as part of the **baseDirs** directory:

```
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() {
    var tsResult =
gulp.src(config.typeScriptSource)
                        .pipe(tsProject());
```

```
    return tsResult.js

.pipe(gulp.dest(config.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:

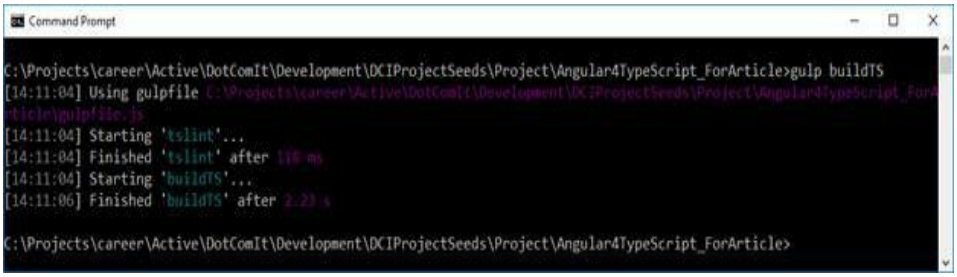
```
var tsResult = gulp.src(config.typeScriptSource)
                   .pipe(sourcemaps.init())
                   .pipe(tsProject());
return tsResult.js

.pipe(sourcemaps.write(config.mapPath))

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

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

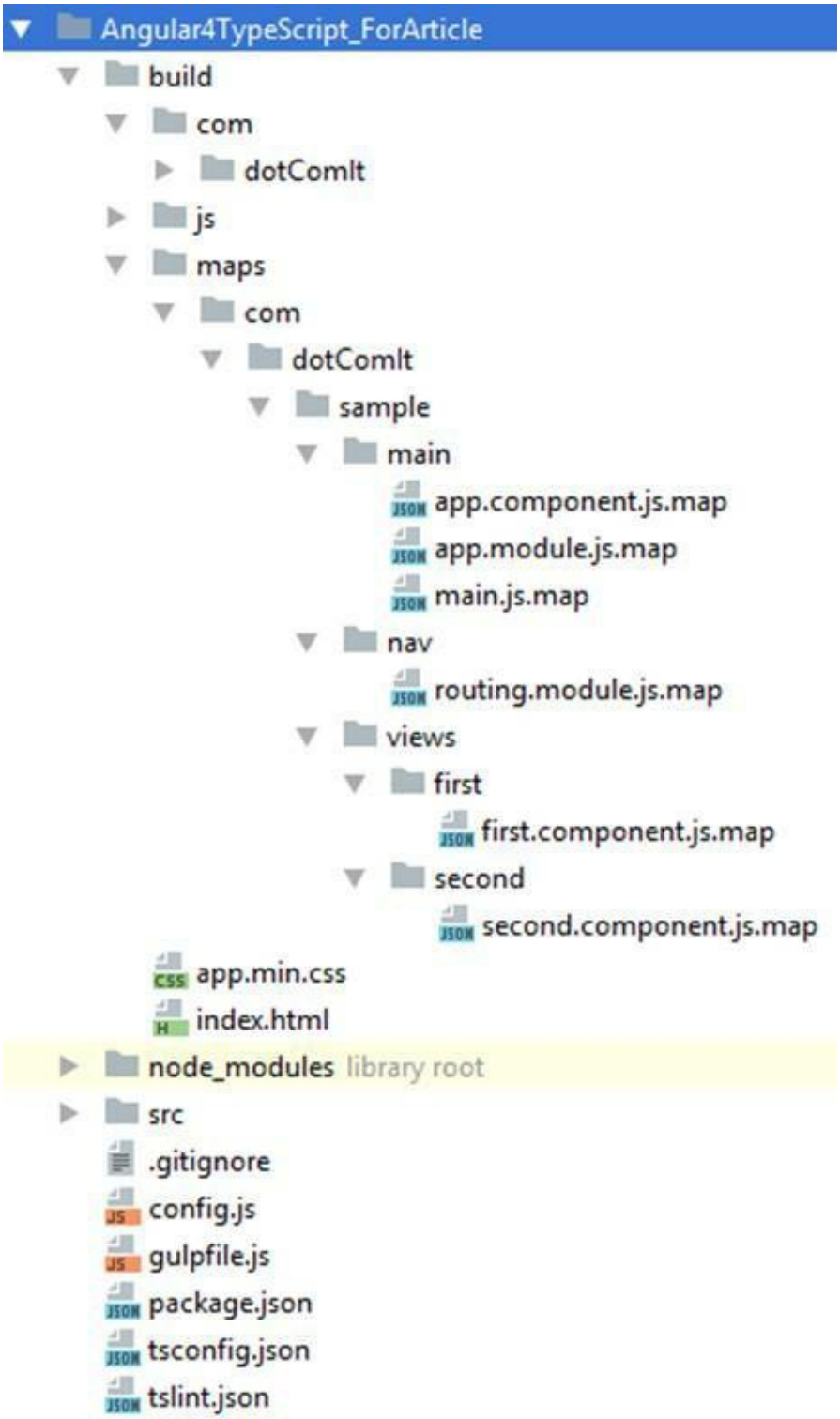
Run the task:

```
gulp buildTS
```

You'll see this:

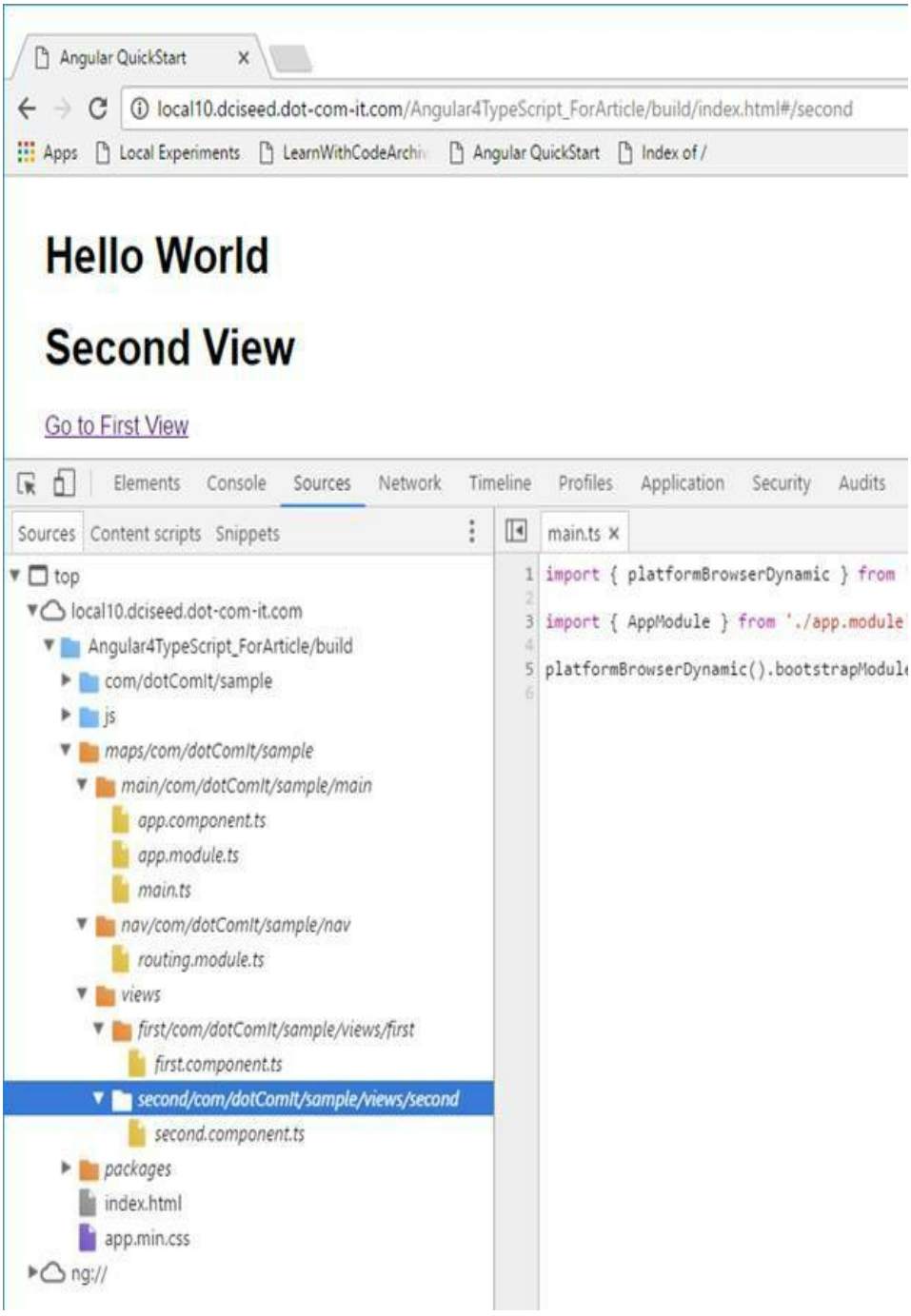


Check the **build** directory:


Angular4TypeScript_ForArticle
build
com
dotComIt
js
maps
com
dotComIt
sample
main
app.component.js.map
app.module.js.map
main.js.map
nav
routing.module.js.map
views
first
first.component.js.map
second
second.component.js.map
app.min.css
index.html
node_modules library root
src
.gitignore
config.js
gulpfile.js
package.json
tsconfig.json
tslint.json

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.

Generate CSS Source Maps

Source maps can also be generated for CSS files. Start by looking at the **processCSS** task:

```
gulp.task('processCSS', function () {
    gulp.src(config.cssSource, { base: '.' })
        .pipe(cleanCSS())
        .pipe(concat(config.cssDestinationFile))
        .pipe(gulp.dest(config.destinationPath))
});
```

Currently it generates no source maps. But, we can add them in. This is the modified body:

```
gulp.src(config.cssSource, { base: '.' })
    .pipe(sourcemaps.init())
    .pipe(cleanCSS())
    .pipe(concat(config.cssDestinationFile))
    .pipe(sourcemaps.write(config.mapPath))
    .pipe(gulp.dest(config.destinationPath))
```

The **init()** method is called right after the source is specified. The **write()** method is called right before the processed files are written to disk.

Let's do the same thing for the Angular CSS files. This is the **copyAngularCSS** task as is:

```
gulp.task('copyAngularCSS', function () {
    gulp.src(config.cssStyleURLsSource)
        .pipe(cleanCSS())

.pipe(gulp.dest(config.destinationPathForCSSStyle
});
```

This task only processes files in the com directory, because it assumes those CSS files will be loaded on demand by Angular's use of the **styleUrls** Component property. Here is the modified task body which creates source maps:

```
gulp.src(config.cssStyleURLsSource)
    .pipe(sourcemaps.init())
    .pipe(cleanCSS())

.pipe(sourcemaps.write(config.destinationPathForC

.pipe(gulp.dest(config.destinationPathForCSSStyle
```

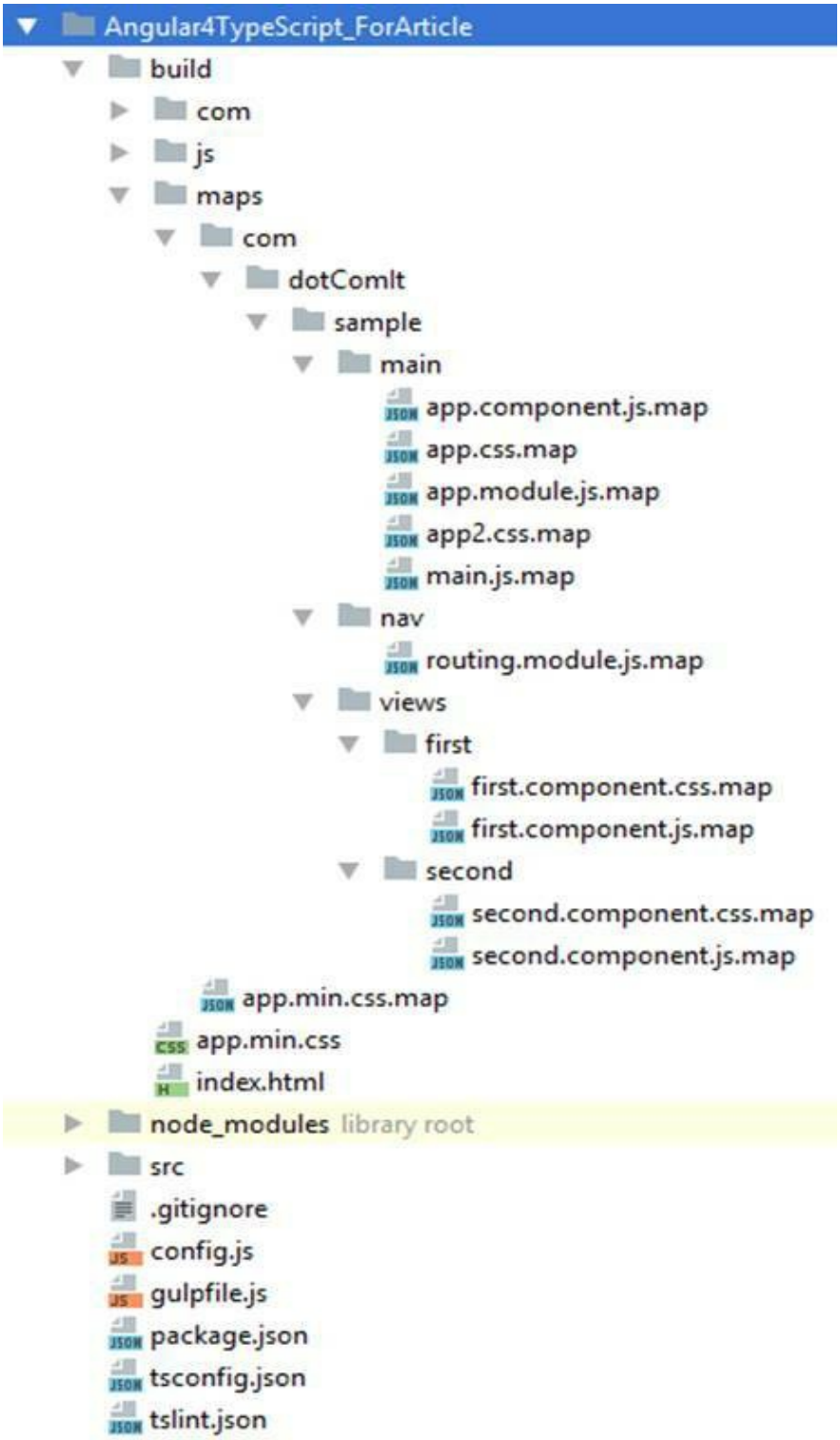
The **init()** method is called first, and the **write()** method is called at the end.

Run both of the CSS tasks with our helper task:

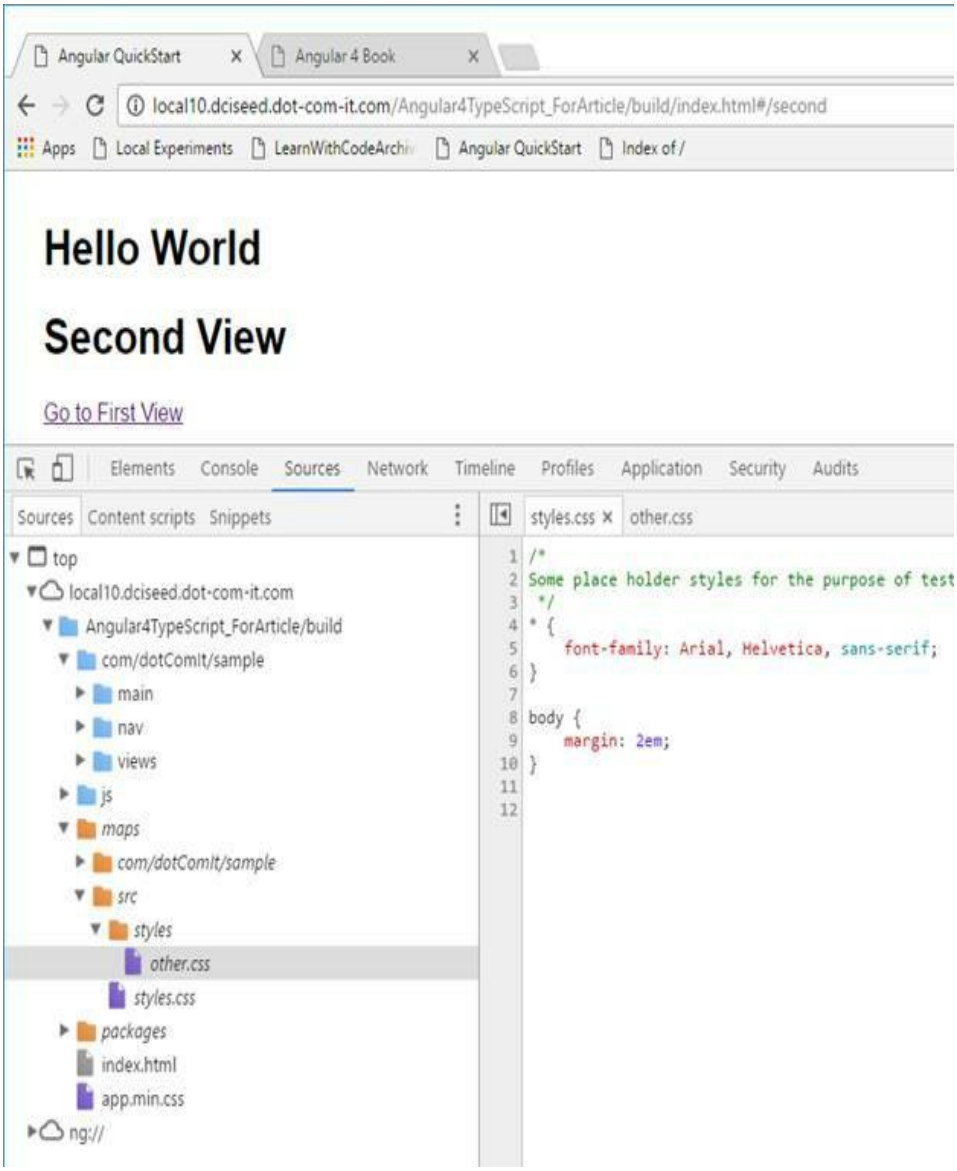
```
gulp buildCSS
```

You'll see a screen like this:

Check out the directory structure:


Angular4TypeScript_ForArticle
build
com
js
maps
com
dotComIt
sample
main
app.component.js.map
app.css.map
app.module.js.map
app2.css.map
main.js.map
nav
routing.module.js.map
views
first
first.component.css.map
first.component.js.map
second
second.component.css.map
second.component.js.map
app.min.css.map
app.min.css
index.html
node_modules library root
src
.gitignore
config.js
gulpfile.js
package.json
tsconfig.json
tslint.json

Source maps work similar whether you're using CSS or JavaScript. Open up a browser and the dev tools to see them:



Source maps for CSS files are more important if you are using less or some other CSS programming technology. For the simple CSS styles in this example, there isn't a huge benefit, but it is nice to have the infrastructure in place.

Minimize 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. We already used **CSS-Clean** to do something similar for CSS files. Let's use UglifyJS to minimize our JavaScript.

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 Build Script

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

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

Now, jump to the **buildTS** task:

```
gulp.task("buildTS", ["tslint"], function() {
    var tsResult =
gulp.src(config.typeScriptSource)
        .pipe(sourcemaps.init())
        .pipe(tsProject());
    return tsResult.js

.pipe(sourcemaps.write(config.mapPath))
        .pipe(gulp.dest(config.destinationPath));
});
```

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

```
    return tsResult.js
        .pipe(uglify())
        .pipe(sourcemaps.write(config.mapPath))
        .pipe(gulp.dest(config.destinationPath));
```

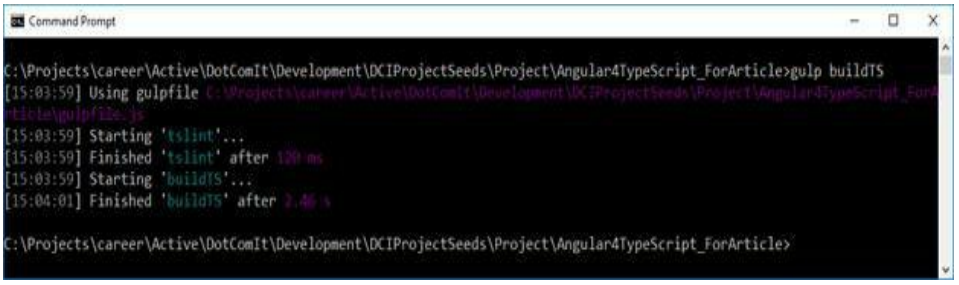
The first **pipe()** on the **tsResult.js** calls the **uglify()** method. We could send in an object if we needed to configure something, but I have found things work acceptably without doing so.

Review the Minimized Code

Run the updated script:

```
gulp buildTS
```

See it run:

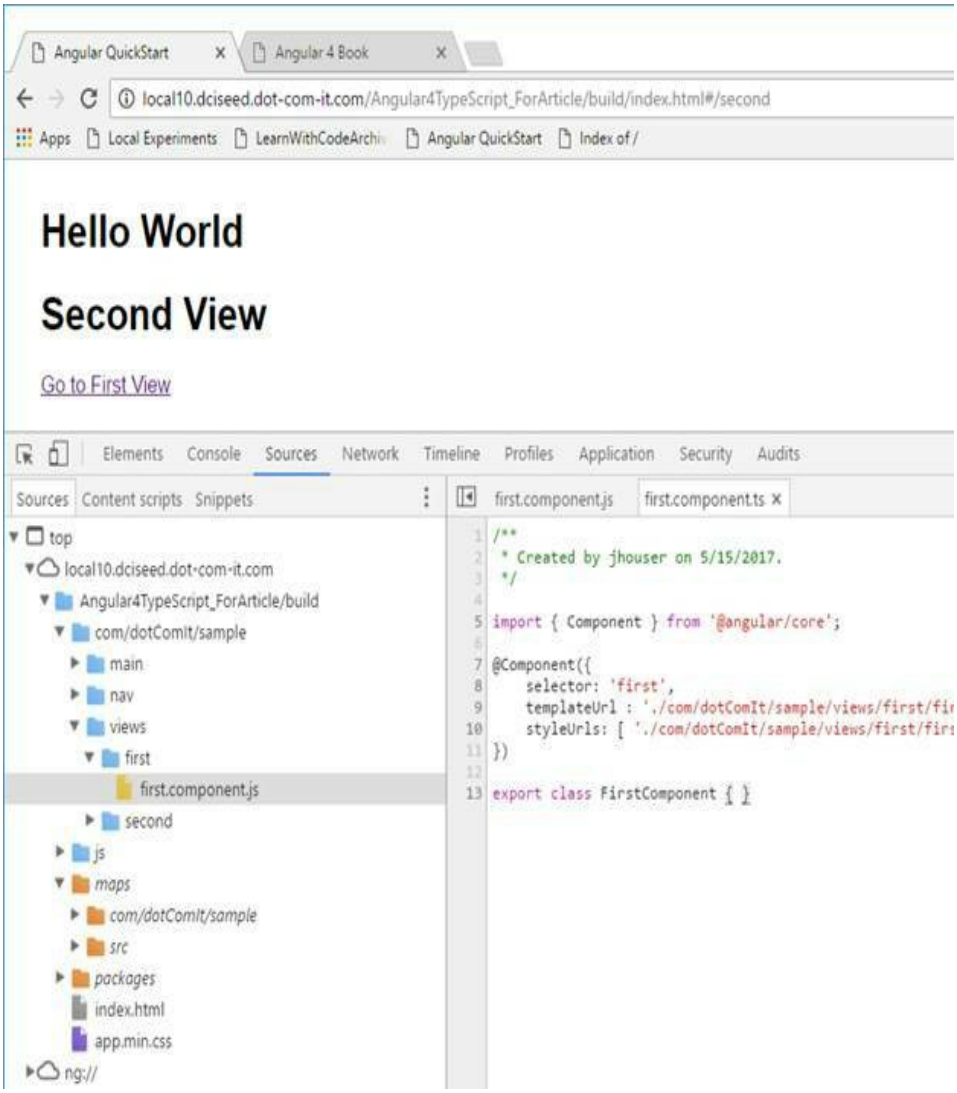


The directory structure will not have changed, but the contents of the custom JS files have. Take a look at the **app.module.js**:

```
"use strict";var
__decorate=this&&this.__decorate||function(e,o,r,
{var p,n=arguments.length,u=n<3?o:null===t?
t=Object.getOwnPropertyDescriptor(o,r):t;if("obje
Reflect&&"function"==typeof
Reflect.decorate)u=Reflect.decorate(e,o,r,t);else
for(var a=e.length-1;a>=0;a--)(p=e[a])&&(u=(n<3?
p(u):n>3?p(o,r,u):p(o,r))||u);return
n>3&&u&&Object.defineProperty(o,r,u),u};Object.de
{value:!0});var
core_1=require("@angular/core"),platform_browser_
browser"),common_1=require("@angular/common"),app
{function e(){}return e}
();AppModule=__decorate([core_1.NgModule({imports
[platform_browser_1.BrowserModule,routing_module_
[app_component_1.AppComponent],providers:
[{provide:common_1.LocationStrategy,useClass:comm
[app_component_1.AppComponent]})],AppModule),expo
//#
sourceMappingURL=../../../../maps/com/dotComIt/sa
```

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 were not present in the original 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.

Write 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:

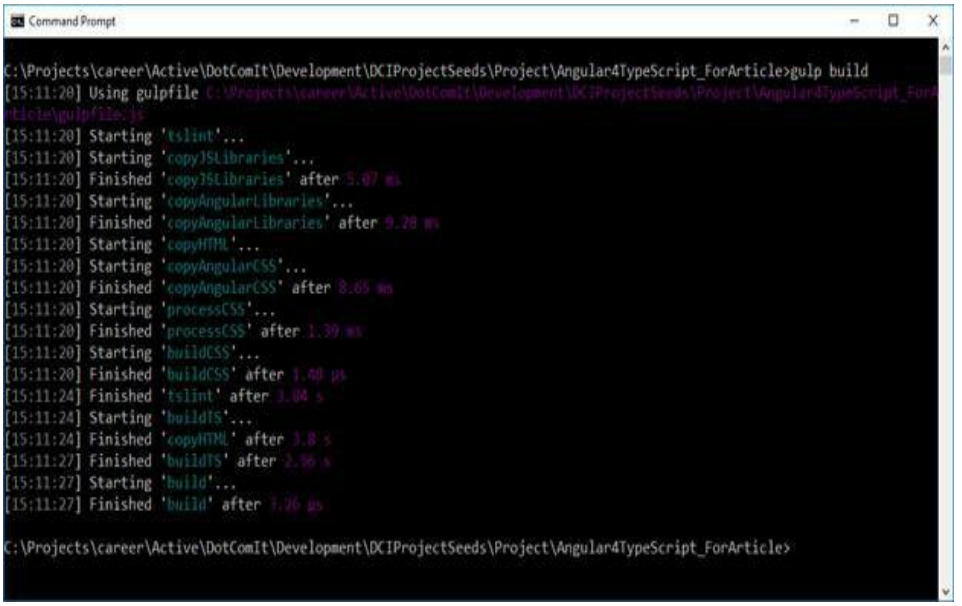
```
gulp.task("build", ['buildTS', 'copyJSLibraries',

'copyAngularLibraries','copyHTML', 'buildCSS']);
```

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 used the same approach when we created the **buildTS** task, where the **tslint** task was executed before the actual **buildTS** task. We also used the same approach when creating **buildCSS** which ran **copyAngularCSS** and **processCSS** tasks. 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 in the **config.js**. I created this as part of the **staticConfig** object:

```
deletePath : [baseDirs.destinationPath + '/**']
```

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

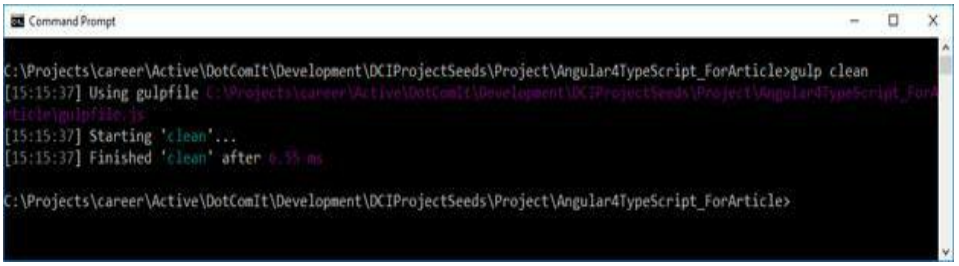
```
gulp.task('clean', function () {
    return del(config.deletePath);
});
```

The task is named clean. It calls the **del()** module with the

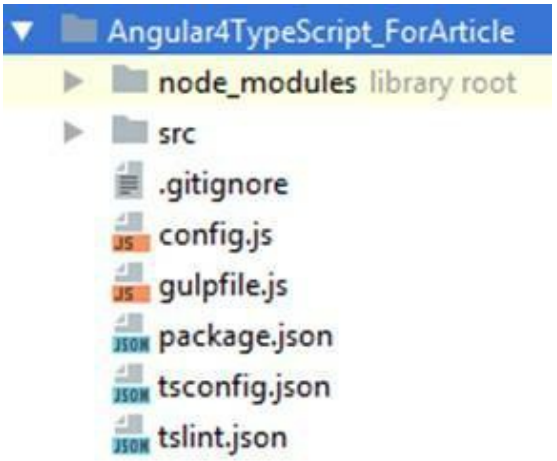
deletePath value. Run the task:

```
gulp clean
```

You'll see this:



Check your project directory:



The **build** directory is noticeably absent.

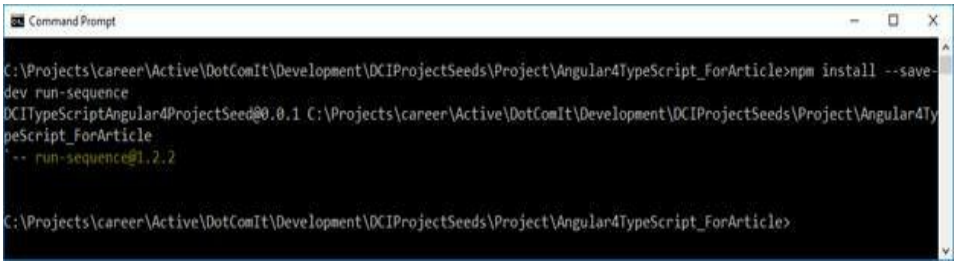
Create a Clean Build

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:

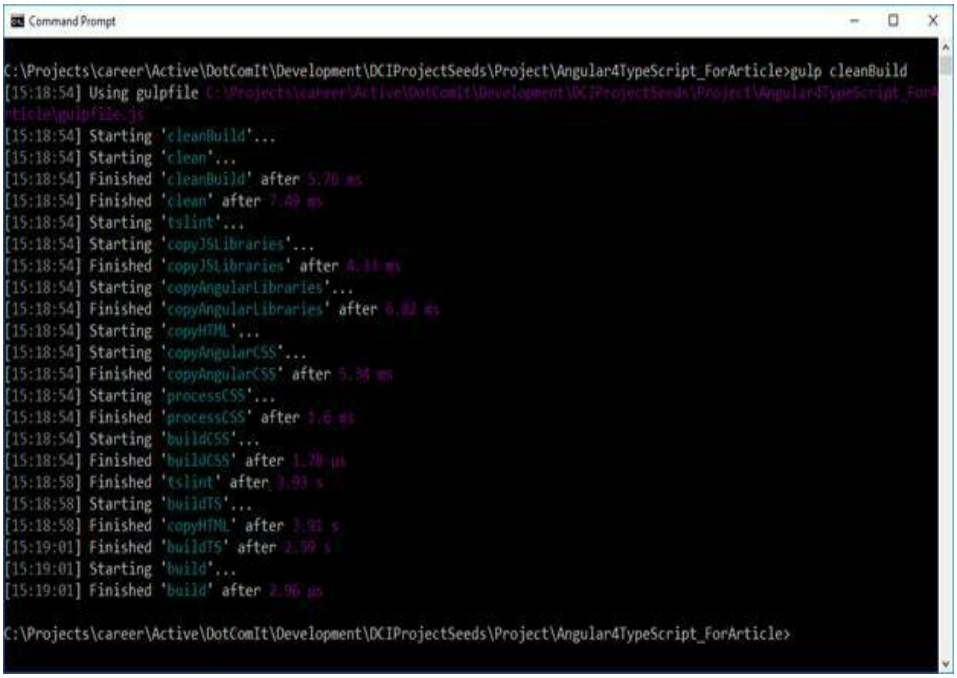
```
gulp.task('cleanBuild', function () {
    runSequence('clean', 'build');
});
```

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:



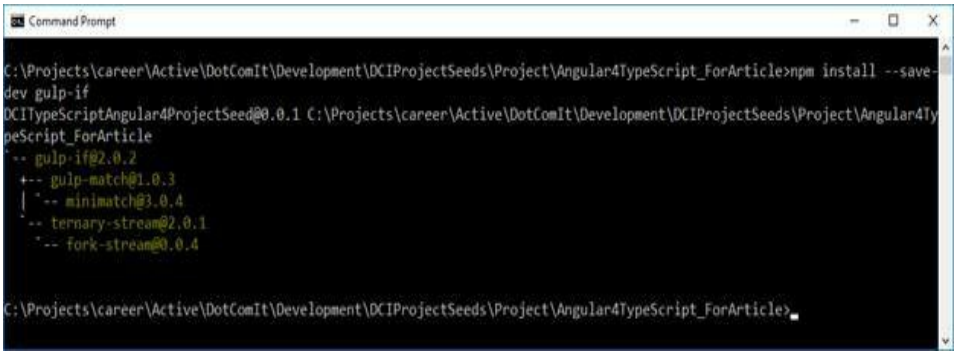
You see that the clean task is run first; and after it is finished the build tasks start. This will delete the build directory and all other relevant files, and then re-generate using the other tasks we wrote about earlier in this chapter.

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**. I put this in the **config.js** file as part of the **staticConfig** object:

```
devMode : true
```

This is the variable we will use to determine whether or not to generate the source maps, and we do not expect users to change it manually. 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() {
    var tsResult =
gulp.src(config.typeScriptSource)
        .pipe(sourcemaps.init())
        .pipe(tsProject());
    return tsResult.js
        .pipe(uglify())
        .pipe(sourcemaps.write(config.mapPath))
        .pipe(gulp.dest(config.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 initialize the source maps.

Also change the **sourcemaps.write()** pipe:

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

We should do the same for the CSS related tasks. First, **processCSS**:

```
gulp.task('processCSS', function () {
    gulp.src(config.cssSource, { base: '.' })

.pipe(gulpIf(config.devMode,sourcemaps.init()))
        .pipe(cleanCSS())
        .pipe(concat(config.cssDestinationFile))

.pipe(gulpIf(config.devMode,sourcemaps.write(conf
        .pipe(gulp.dest(config.destinationPath))
});
```

And then **copyAngularCSS**:

```
gulp.task('copyAngularCSS', function () {
  gulp.src(config.cssStyleURLsSource)

.pipe(gulpIf(config.devMode,sourcemaps.init()))
      .pipe(cleanCSS())

.pipe(gulpIf(config.devMode,sourcemaps.write(

config.destinationPathForCSSStyleURLMaps)))

.pipe(gulp.dest(config.destinationPathForCSSStyle
});
```

The edits to the two CSS tasks mirror exactly what we did for the **buildTS** task.

Now, 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:

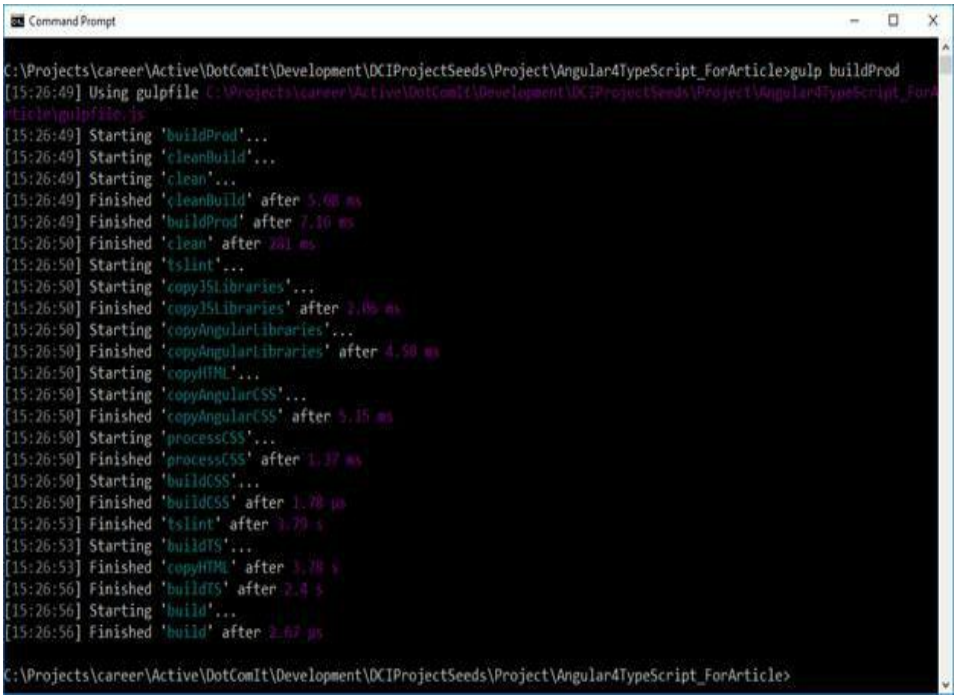
```
gulp.task('buildProd', function(){
    devMode = false;
    gulp.start('cleanBuild')
});
```

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.

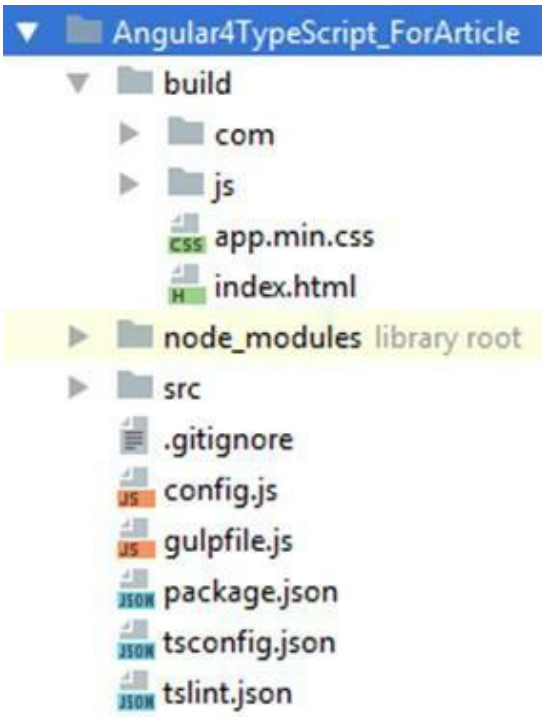
Run the task:

```
gulp buildProd
```

You should see this:



Review the build directory:



You'll notice that the maps directory is missing; meaning that the source maps were successfully bypassed when running the **buildProd**.

Watch Directories for Code Changes

There was one more thing I wanted to touch on. 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 that before our function is executed. 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:

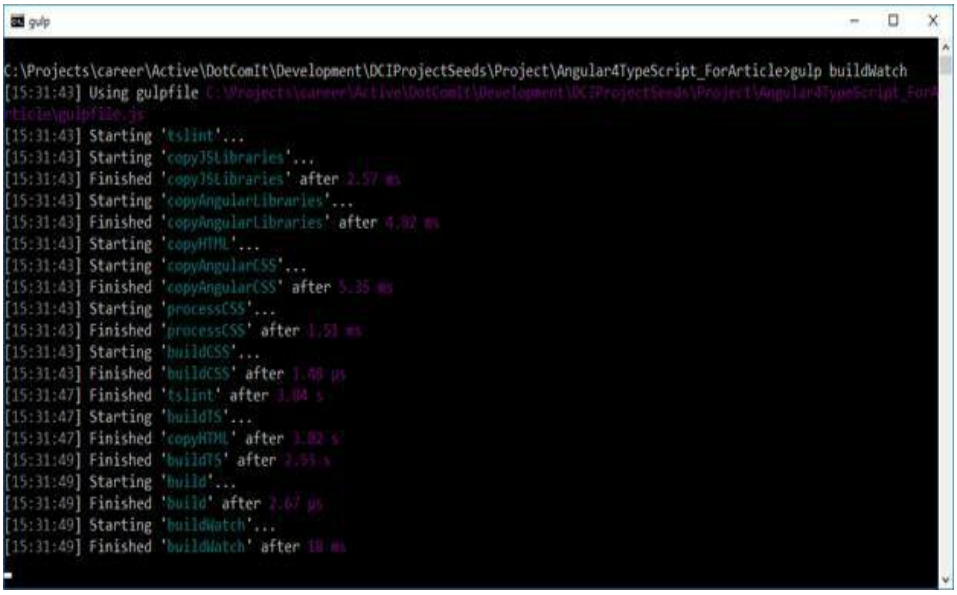
```
gulp.watch(config.typeScriptSource,
['buildTS']).on('change', function(event){
    console.log('File Path' + event.path);
});
```

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
```

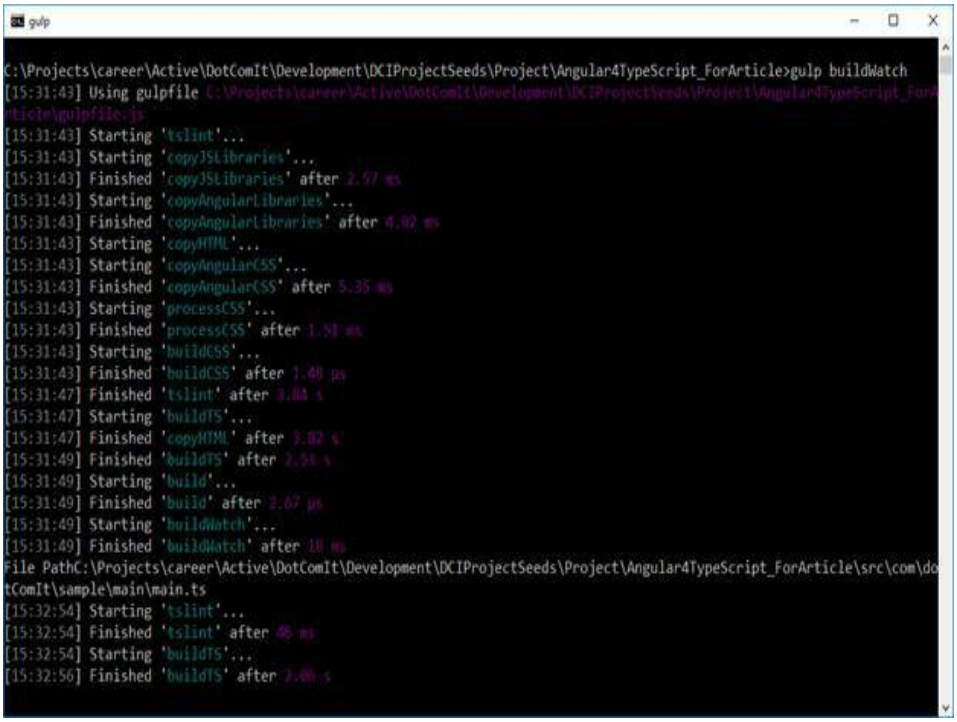
You should see something like this:



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, JS, and CSS libraries:

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

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

```
gulp.watch(config.cssStyleURLsSource,
['copyAngularCSS']).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 or 4 with TypeScript, the build process became much more important since a compile process is needed before you can test or run your code.

Chapter 2: Using the Angular CLI

When I started writing my book on Angular, there were not a lot of options available for seed projects. I ended up creating my own, as documented in Chapter 1, and used in the main book. By the time I was ready to publish, the Angular CLI had taken over the market. Now, even Angular's official quick start contains samples that use the Angular CLI. This article will teach you all about it.

What is Angular CLI?

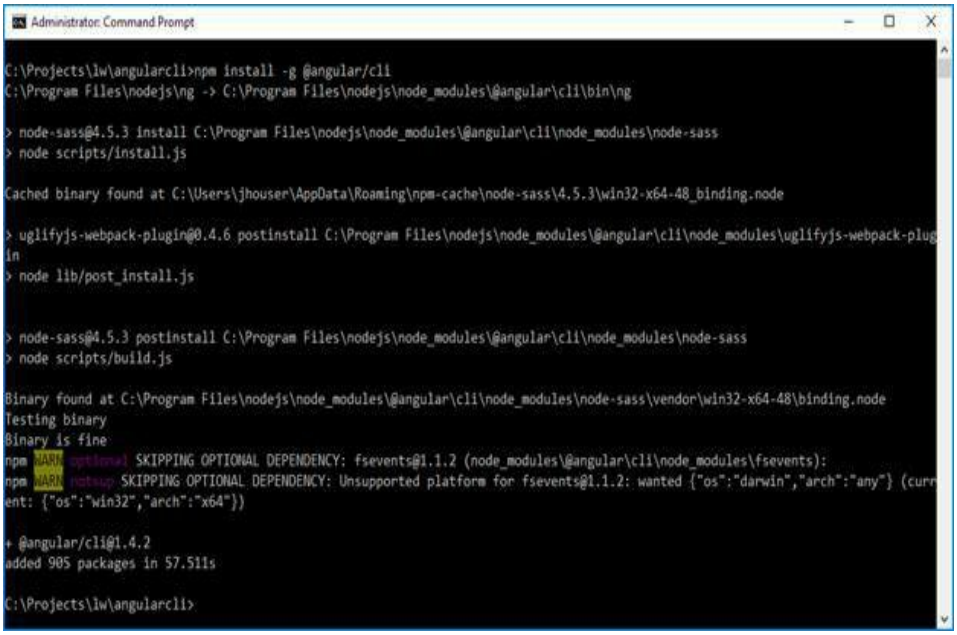
The Angular CLI is a command line interface for developing Angular applications. It is built on top of NodeJS, and contains workflow scripts to create an Angular project, add components, and run unit tests. It handles the complication under the hood including compiling TypeScript to JavaScript, creating production ready builds, and using other components. It even includes a built-in web server which will automatically reload the browser when you change your code. The Angular CLI has become the standard build tool for building Angular applications.

Install Angular CLI

The first thing you need to do is install the Angular CLI project. You can do this with this node command:

```
npm -g install @angular/cli
```

I recommend installing it globally, which is what the -g flag does. You should see a screen like this:



Now, the library is installed, and you can use it.

Setup a Default Project

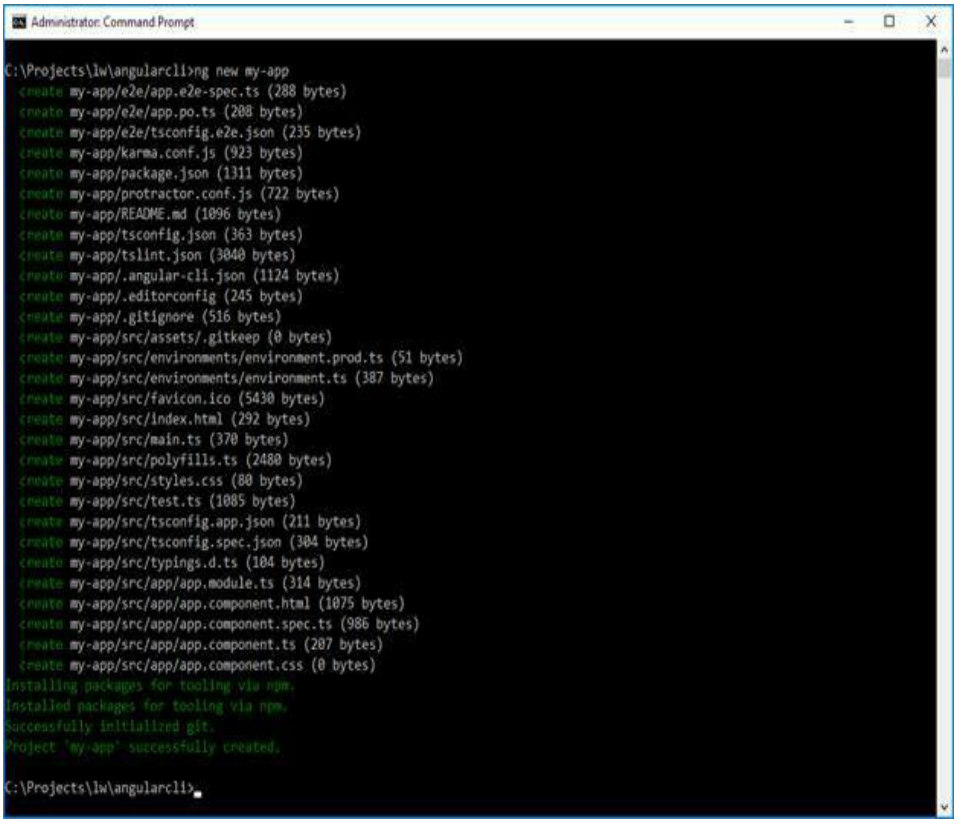
This section will teach you about the default project seed that allows you to build a simple project with Angular CLI.

Install the Seed

You can create a new project like this:

```
ng new my-app
```

The **ng** command refers to the Angular CLI project. The new command tells it you want to create a new project. The **my-app** argument tells the script to put all the new project files inside the **my-app** directory:

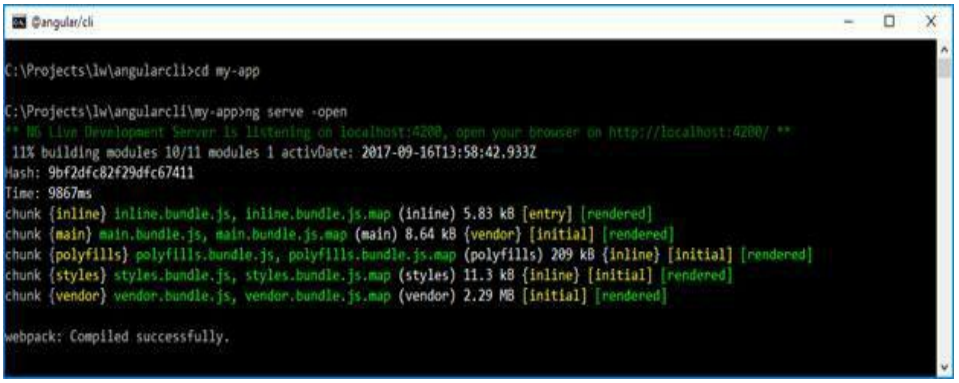


This will create a sample project in the **my-app** directory. We'll

expand the files and directory structure a bit, but first let's compile the sample project. Change to the **my-app** directory, then run this:

```
ng serve –open
```

You'll see something like this:



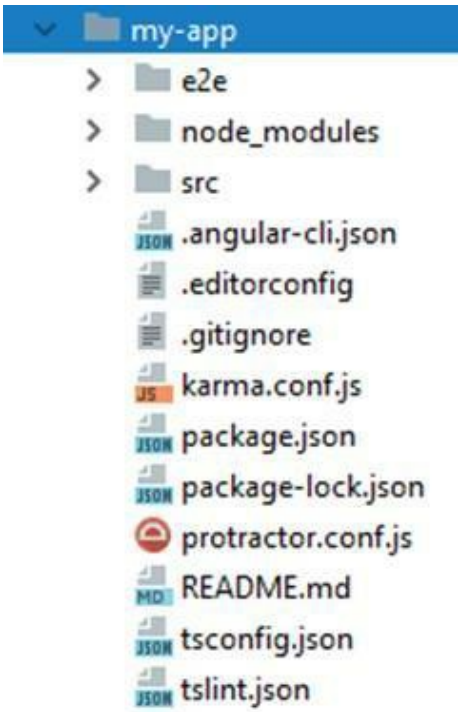
The **ng-serve** command will compile the app, start a web server, then launch your default web browser:


Jeffry
MyApp
localhost:4200
Welcome to app!
Here are some links to help you start:
Tour of Heroes
CLI Documentation
Angular blog

Congratulations, you just created your first Angular app with the Angular CLI.

Review the Project Seed

Before moving on, let's examine the **my-app** directory to see what Angular CLI created as a project seed:



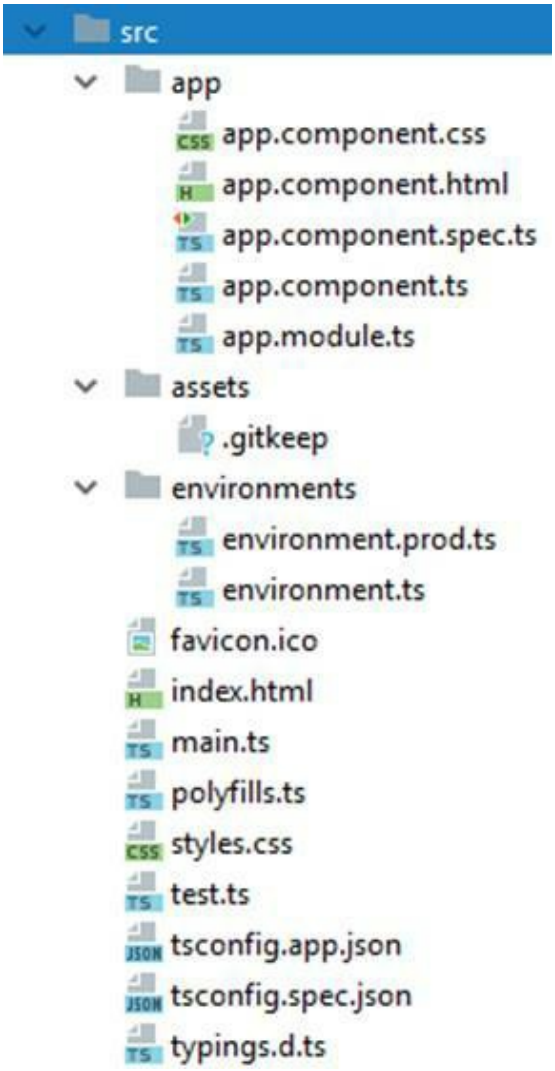
These are the relevant directories and files:

- **e2e**: This directory contains end to end tests, something beyond the scope of this book.
- **node_modules**: This directory contains all the installed NodeJS modules used to build the application. Angular and TypeScript are two such libraries.
- **src**: This contains the application source, you'll spend most of your time writing code here.
- **.angular-cli.json**: This contains the configuration for the Angular CLI project. You can this wiki article for more

information.

- **.editorconfig**: This is a default IDE Configuration file, and is beyond the scope of this article. You can check out editorconfig.com for more information.
- **.gitignore**: This is a git version control file and is used to tell your git tool to ignore certain files. This is beyond the scope of this chapter.
- **karma.conf.js**: Karma is a unit test runner, and this file is its configuration.
- **package.json**: This file is Node's default configuration, including all libraries.
- **package-lock.json**: This file is a configuration log generated when NodeJS modifies the **node_modules** or **package.json**. Read more about it. For the purposes of this chapter, it can be ignored.
- **protractor.conf.js**: Protractor is an end to end testing tool used by the Angular CLI.
- **README.md**: The **README.md** file is intended as a basic documentation file of your project, and GitHub uses them to create fancy page descriptions with links and general rich text. This is beyond the scope of this article.
- **tsconfig.json**: This file contains your TypeScript configuration and many IDEs will use this to show you compile errors as you write.
- **tslint.json**: This is a lint configuration file. Linting is a process of reviewing code against a standard and point out inconsistencies. TSLint and CodeAnalyzer are used for linting by the Angular CLI.

Let's look at the **src** directory in some more detail:



This is what all of this means:

- **app**: This directory contains the main application files, most specifically the **app.module.ts** which defines the project's module and the app's main component at **app.component.ts**. It includes supporting files for the component including an HTML file as a template, a CSS file for styles, and a spec file for tests. My one complaint about this structure is that I prefer to store my tests independent of my source code, but if you're using Angular CLI you should follow the Angular CLI way.

- **assets**: This folder is supposed to contain any binary assets, such as images.
- **environments**: This folder contains a configuration file for each of your destination environments. By default, it comes with one for production deployments and one for development deployments. Out of the box, this just specifies a production Boolean, which is true for production or false for development. You may want to put other values in here, such as service endpoints.
- **favicon.ico**: This is an image file that represents the browser icon. Since I primarily work on internal apps, most clients don't care about this and I'd just delete it.
- **index.html**: This is the main index page. Most likely you won't need to edit this, as Angular CLI will automatically add the JS and CSS tags as needed.
- **main.ts**: This represents the main application file for your application. It will load the app's main module from the app directory, which will in turn load the app's main component. I don't like that this is in the root directory, but it does work.
- **polyfill.ts**: This is a bunch of libraries that help an Angular app work in as many browsers as possible, even if the browser doesn't support the latest and greatest intricacies of JavaScript.
- **styles.css**: This is an empty CSS file for your applications main app styles. I'd prefer this was not in the root directory, especially since many applications I've worked on use multiple style sheets.
- **test.ts**: This file is the main entry point for your unit tests.
- **tsconfigapp.json**: The TypeScript compiler configuration. It inherits from the **tsconfig.json** file in the root directory, but it an easy way for you to configure settings on an app-by-app basis.

- **tsconfig.spec.json**: The TypeScript compiler configuration for the Unit Testing Code.
- **typings.d.ts**: This contains TypeScript definitions, and are beyond the scope of this book.

The bulk of you time writing applications will be in the source directory, but it is good to have a base of everything that your building on top of.

Modify Angular Components

A cool thing about the AngularCLI is that it will update the browser on the fly as you change code. First, make sure your server is running use this command:

```
ng server -open
```

We used this command earlier in this chapter.

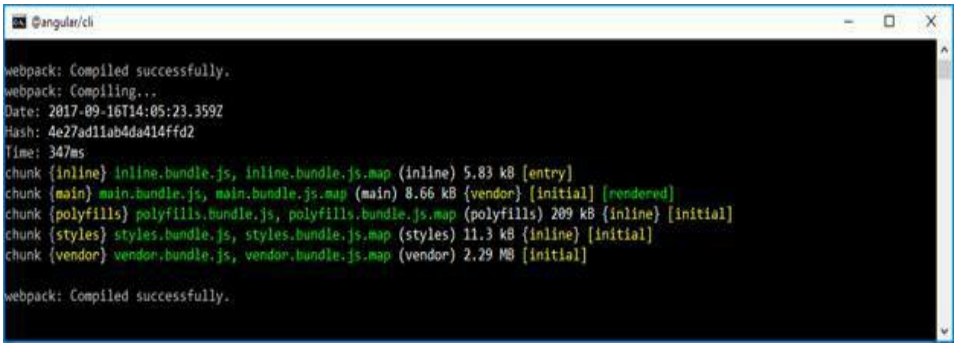
Now, open the **src/app/app.component.ts** file and you'll see something like this:

```
export class AppComponent {
  title = 'app';
}
```

The title is the description displayed inside the **app.component.html** file. Change the text to something else:

```
export class AppComponent {
  title = 'Learn With, Readers';
}
```

Save the file and the seed automatically rebuilds the app and reloads the browser automatically:



The browser will also update:


Jeffry
MyApp
localhost:4200
Welcome to Learn With, Readers!
Here are some links to help you start:
Tour of Heroes
CLI Documentation
Angular blog

The browser’s auto update is a nice feature to have and can save you a lot of debugging time compared to manually recompiling your application.

Generate Elements Automatically

One aspect of Angular CLI is that it be used to generate stubs for many TypeScript elements. I want to give a few examples. We're going to show two examples, one for a TypeScript Class and one for an Angular Component, but Angular CLI can generate more than just that.

Generate a Class

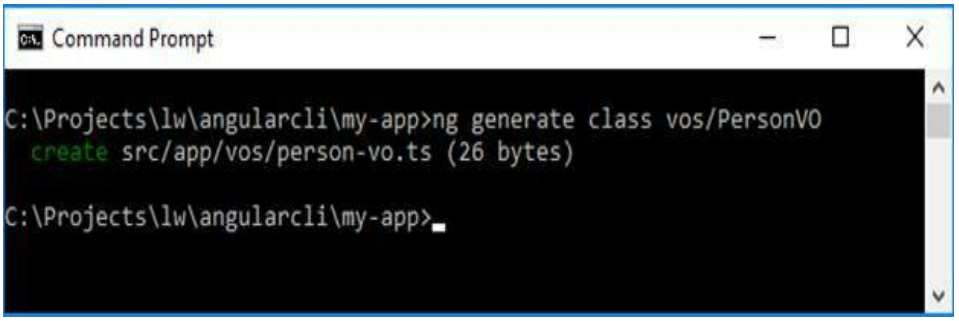
Angular CLI uses the generate command to create a value object class. It is used in this format:

```
ng generate class dir/classname
```

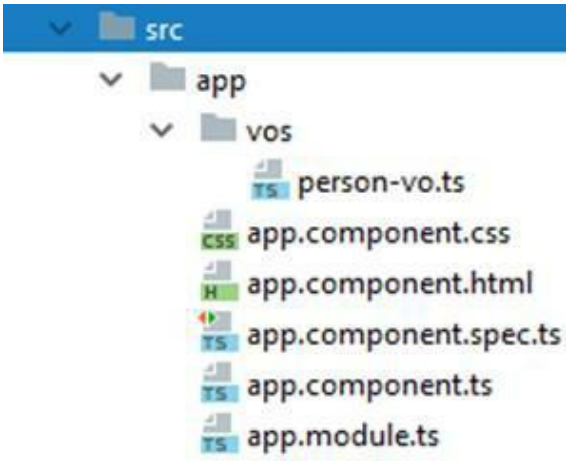
The class is always created relative to the **src/app** directory. Let's create a **person** class:

```
ng generate class vos\PersonVO
```

This will create a new class, **PersonVO**, inside the **src/app/vos** folder. The **VO** in the class name and directory represents a value object which is a design pattern for an object that represents a single entity. In this case, the entity is a person.



Look at the directory structure to see the new class:



You'll see the new directory, **vos**, and a **person-vo.ts** file inside it. Open up the file:

```
export class PersonVo {
}
```

It is a simple class definition with no body or other details yet. Let's add a first name and last name property:

```
export class PersonVo {
  firstName : string;
  lastName : string;
}
```

We'll use this a bit later as an input into a new component.

One thing to notice with this generation is that the file name, **person-vo,** is not the same as the class name, **PersonVO**. Normally I'd mirror the file name with the class name, but the Angular CLI automatically put the file name into all lowercase, and put a dash where the case changed, between the "n" and the "v". Angular CLI is using a different naming convention than what I'm used to.

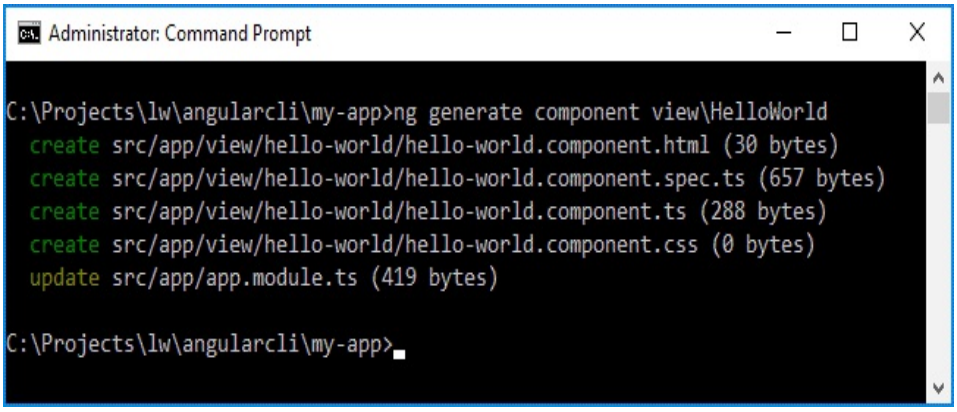
Generate a Component

Let's use Angular CLI to generate a component. Run this command:

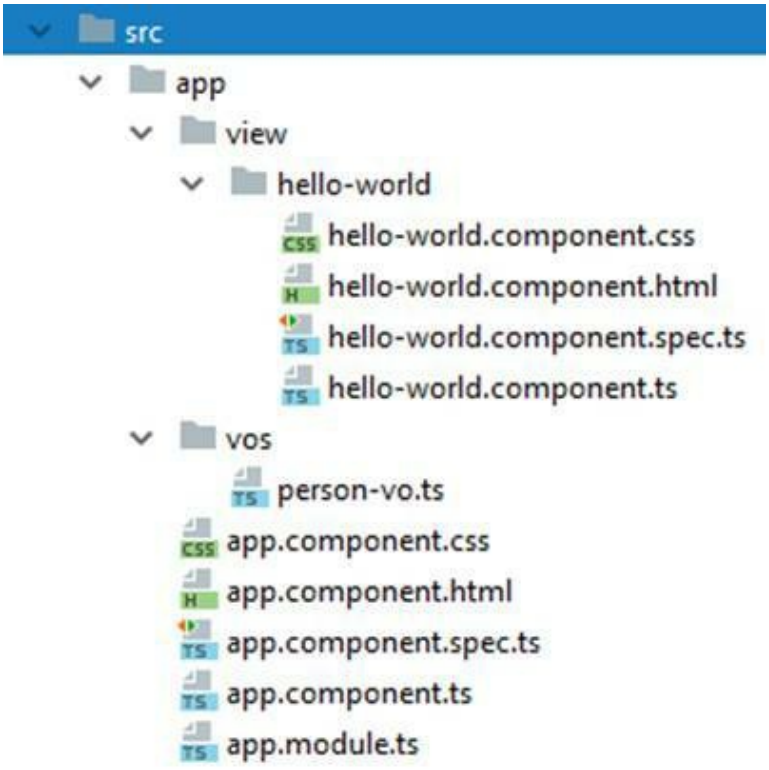
```
ng generate component view/HelloWorld
```

The syntax is similar for generating a class. You'll see something like

this:



Let's see what it did for us:



It put the **HelloWorld** component in the **view/hello-world directory**. I like that it automatically put all the relevant component files in the same directory. I like that it automatically split up the CSS, HTML, spec, and class files. As with class generation, it named

the files differently than the actual class name, putting everything into lower case, with a dash when the classes name changed. It also added the component postfix to the file names. I don't mind this, though. A lot of these things can be configured via command line arguments, but I decided to stick to the defaults for this.

Let's look at files one by one. The CSS file is empty, waiting for us to add some custom styles. The HTML component adds some simple text:

```
<p>
  hello-world works!
</p>
```

In most cases, you'll be replacing that with something more complex. The component TypeScript file contains this:

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

@Component({
  selector: 'app-hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.css']
})
export class HelloWorldComponent implements
OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

I like this setup, and having Angular CLI generate it is a lot easier than writing the code from scratch. It defines the **@Component** metadata, creates the class, adds the constructor and **ngOnInit()** methods. In most components, you're going to want this.

The spec file is a sample unit testing file, which I'm going to skip

over for the purposes of this article. A cool thing to notice is the **app.module.ts** file:

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

import { AppComponent } from './app.component';
import { HelloWorldComponent } from './view/hello-
world/hello-world.component';

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

It was automatically updated to import the **HellowWorldComponent** and add it to the declarations. That is a huge time saver, especially since I often forget to properly add my new components here.

Working with our New Component and Class

Let's make a few changes to see how we can make our new component and class work together. First, open the **hello-world-component.ts** file. We'll give this an input property of type **PersonVo**.

Add an import statement for the **PersonVo** class:

```
import {PersonVo} from "../../vos/person-vo";
```

Now, add an **input** variable to the class:

```
@Input()
person : PersonVo;
```

Don't forget to import the **Input** class:

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

Now, open up the **hello-world.component.html** file:

```
<p>
  Hello {{person.firstName}}
{{person.lastName}}!
</p>
```

We use this file to say hello to our person.

In the **app.component.ts**, we're going to create a **Person** Object. First, import the class:

```
import {PersonVo} from "./vos/person-vo";
```

Then inside the class, create a **person** variable:

```
person :PersonVo;
```

And add a constructor to create the **person** variable:

```
constructor() {
  this.person = new PersonVo;
  this.person.firstName = "LearnWith";
  this.person.lastName = "Users";
}
```

Delete everything in the **app.component.html** file and replace it with this:

```
<div style="text-align:center">
  <h1>
    <app-hello-world [person]="person"></app-
hello-world>
  </h1>
</div>
```

Using the **app-hello-word** as a tag, we inject that component's view into our current view. Using the square brackets, we pass it our new value. If you're running the app, check the browser to see our updates:

The lesson here is that even though Angular CLI can give you a jump start on creating classes and components, you're still going to have write your own business logic to flesh out the application.

Generating Other Things

The Angular CLI can generate more than just TypeScript classes and Angular components. Here are a few other things you can create with **ng-generate** that are TypeScript elements, but not part of Angular:

- enum**:** This generates an Enum. Enum's are a special data type in Angular, like a cross between a class, an array and a dictionary.
- interface**:** This generates an interface, which is a class definition without an implementation.

Here are a few things that Angular CLI will generate which are Angular specific:

- directive: A directive is a way to modify the structure of a page, such as with **NgIf** or to add an attribute to an HTML

DOM element.

- guard**:** A guard relates to Angular routing.
- module**:** This generates an Angular module.
- pipe**:** A pipe is customized data filter in Angular
- service: A service is used to share data or functionality across multiple components.

I decided to expand on things I'd use most commonly, however you can see there is a lot in that Angular CLI can help you generate.

Install and Use ng-bootstrap

A part of building any Angular application is going to be relying on third party libraries. I'm going to show you how to use ng-bootstrap with Angular CLI.

Install Dependencies

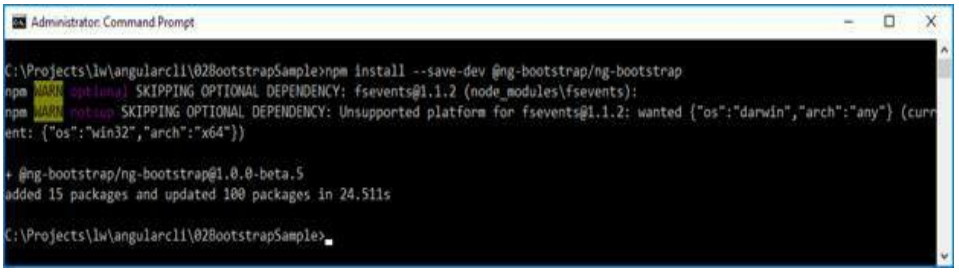
Let's go back to the default example and expand it to add a default library. First install it:

```
ng new 02BootstrapSample
```

This will create a new directory named **02BootstrapSample** that represents the default Angular CLI project seed. Switch to the **02BootstrapSample** directory and then use npm to install the ng-bootstrap library:

```
npm install –save-dev @ng-bootstrap/ng-bootstrap
```

You'll see something like this:



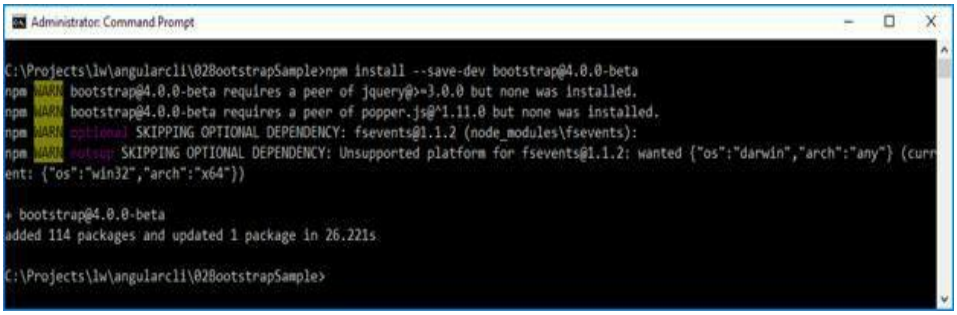
This installs the ng bootstrap library. However, we also need to load the Bootstrap CSS. We could do this using a CDN and add this to the **index.html** file:

```
<link rel="stylesheet"

href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.min.css" />
```

Or we could install regular bootstrap using an npm package:

```
npm install --save-dev bootstrap@4.0.0-beta
```

You'll see something like this:



Don't worry about the required peer dependencies, we don't need them since we're using ng-bootstrap.

We do need to tell Angular CLI to copy CSS files into our project. Open up the **angular-cli.json** file. Look for the **apps/styles** section. It should look like this:

```
"styles": [
  "styles.css"
],
```

Just add the bootstrap CSS to the styles array:

```
"styles": [
  "styles.css",

"../node_modules/bootstrap/dist/css/bootstrap.css
],
```

Now when we build the application the bootstrap styles should be loaded.

Use Bootstrap

For simplicity I'm going to use the Bootstrap alert. First, open up the **app.module.ts**. Add an import for the **NgbModule**. This is the independent module that Bootstrap loads:

```
import {NgbModule} from '@ng-bootstrap/ng-
bootstrap';
```

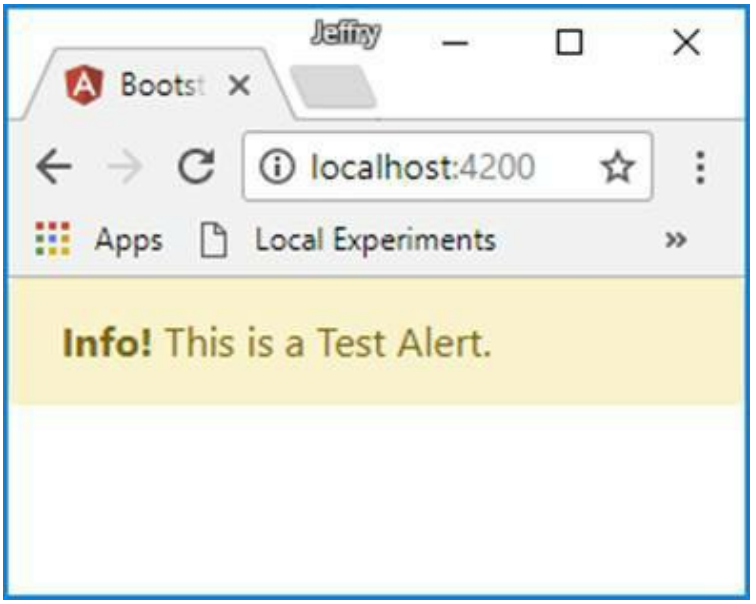
Then be sure to import that module as part of the **@NgModule** metadata:

```
imports: [
  BrowserModule,
  NgbModule.forRoot()
],
```

Now open up the **app.component.html** file:

```
<p>
  <ngb-alert [dismissible]="false">
    <strong>Info!</strong> This is a Test Alert.
  </ngb-alert>
</p>
```

Rerun the app with the **ng serve** command and look at it in the browser:



You've successfully used Bootstrap as part of your application.

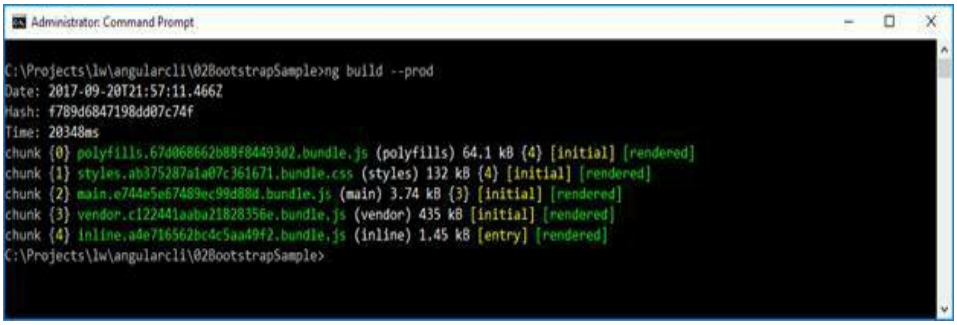
Create a Production Build

You've probably noticed that when running **ng serve**, you cannot find the generated files. Under the hood, the files are only generated in memory and not saved to disk. So, what do you do when you need to deploy the files? Angular CLI provides a special command, **ng build**.

Run the build:

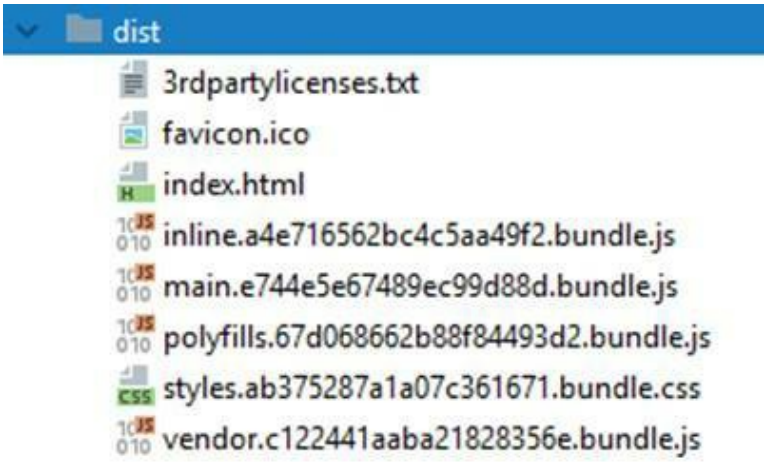
```
ng build –prod
```

You'll see something like this:



This will generate the files that you can then deploy to a web site of your choice.

The generated files will reside in the **dist** directory:



Creating a production build minimizes the files with UglifyJS. These are the files included:

- **3rdpartylicenses.txt**: This file contains license information. It is not needed for your application to run, but you probably want to keep it around.
- **favicon.ico**: this is the default favicon that came with seed app. You can specify this, and other assets in the **angular-cli.json** config file.
- **index.html**: This is the main index file.
- **inline.*longstring*.bundle.js**: Angular CLI uses Webpack under the hood. Webpack is a build tool and this file is a web pack loader file.
- **main.*longstring*.bundle.js**: This contains the minimized version of all your TypeScript code, in minimized JavaScript format.
- **polyfills.*longstring*.bundle.js**: This contains the minimized version of the polyfill library, which is part of the default project.
- **styles.*longstring*.bundle.css**: This file contains the minimized and concatenated CSS files.

- **vendor.*longstring*.bundle.js**: This file contains the build of the Angular libraries.

Just move the files from the **dist** into a web directory and your app is good to go.

Covert the Learn With Task Manager to Bootstrap

I thought it'd be beneficial to show you can convert an existing project to the Angular CLI way, so I took the application you built in the LearnWith book and converted it. This section will show you how.

Setup a New Project

First you need to setup a new project. We've done this a few times already:

```
ng new learnwith
cd learnwith
ng serve -open
```

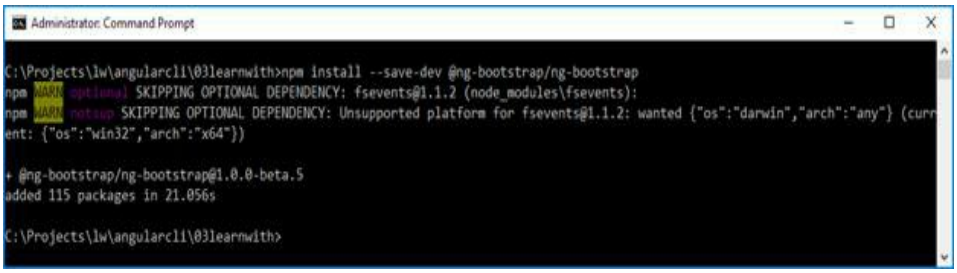
This will setup a default project with the Angular CLI Seed and prove that it works.

Install Dependencies

Over the course of the LearnWith book, we use a lot of third party libraries to accomplish things. Let's install those into our LearnWith project. Start with ng-bootstrap:

```
npm install -save-dev @ng-bootstrap/ng-bootstrap
```

You'll see something like this:

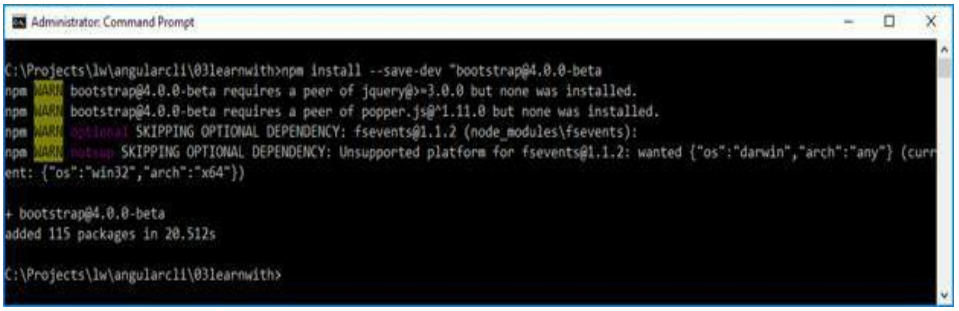


This mirrors the ng-bootstrap install we did in the previous section. From ng-bootstrap we use a lot of TypeScript libraries

We also want to install bootstrap, so we can get it's CSS:

```
npm install --save-dev "bootstrap@4.0.0-beta
```

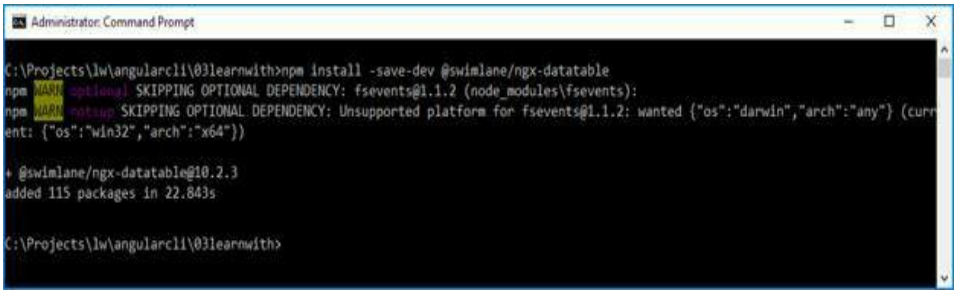
You'll see something like this:



We also need our custom DataGrid library:

```
npm install –save-dev @swimlane/ngx-
datatable@8.0.1
```

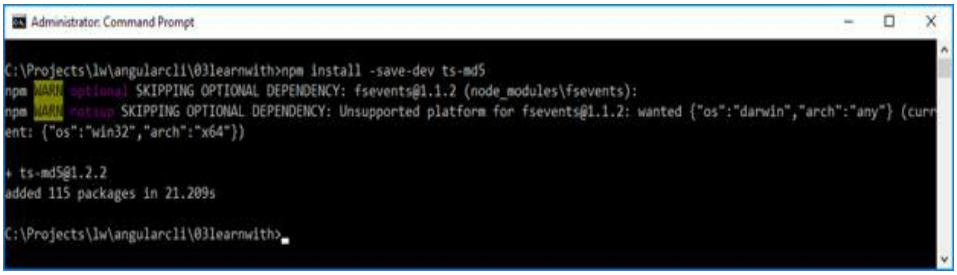
Take a look here:



So far so good. The last library we need to install is the hash library used for sending passwords:

```
npm install –save-dev ts-md5
```

You'll see something like this:



Now all our extra dependencies are loaded into the application and ready to be imported into our Angular components.

Setup Library Assets

Even though the TypeScript code from our libraries is ready to be used, we still need to tell Angular CLI how to get non-TypeScript assets, such as CSS and font files. Open up the **.angular-cli.json** file. First, we need to tell it where to find the CSS for bootstrap. You already know how to do this from the previous section:

```
"styles": [
  "styles.css",

"../node_modules/bootstrap/dist/css/bootstrap.css
],
```

Find the styles section and add the location for the **bootstrap.css**. The location is relative to the source directory, not the project root.

Our DatePicker's are using an image asset for the calendar icon. Copy that from the old project into the new source. I copied the **img** directory in the new **src** directory. How find the assets section of the **angular-cli.json**:

```
"assets": [
  "img"
],
```

Angular CLI will automatically copy all elements inside the **img** directory to the **dist** when we make builds. While I was playing with assets I also deleted the **asset** folder which was installed into the **src** directory as part of the default project.

Since we're dealing with assets, we need to copy the font files used

by NGX-Datatable as part of the new **dist** folder. In the old approach we were copying the files with a Glob:

```
node_modules/@swimlane/ngx-
datatable/release/assets/fonts/*.*
```

We're going to use a similar Glob here, as part of the assets array:

```
{
  "glob": "**/*",
  "input": "../node_modules/@swimlane/ngx-
datatable/release/assets/fonts",
  "output": "./fonts"
}
```

Instead of a string, like we used before, we're not using an object. It has three elements, the glob, the **input**, and the **output**. The glob property represents the file mask. The **input** property is relative to the source directory and tells us the directory where we want to look for files. The **output** property is relative to the **dist** directory and tells us where to move the files to. Adding this to the **assets** array tells Angular CLI to look in the fonts directory of the ngx-datatable install and copy all those files to the fonts directory of the final build location.

We also need to tell Angular CLI to look at ngx-datatable styles, so they can be picked up. In the old system we were using a glob for this:

```
'node_modules/@swimlane/ngx-
datatable/release/**/*.css',
'!node_modules/@swimlane/ngx-
datatable/release/**/app.css'
```

This said to get all the CSS files in the ngx-datatable release, but ignore the **app.css**. This was because the **app.css** was styles for the ngx-datatable sample app, not for the actual component. Unfortunately, Angular CLI does not support globs for styles because the order in which styles are included is important and when using a glob you have no control.

We can get around this by listing all the styles one by one:

```
"../node_modules/@swimlane/ngx-
datatable/release/assets/icons.css",
"../node_modules/@swimlane/ngx-
datatable/release/themes/material.css"
```

Add those two lines to the styles array in the **.angular-cli.json** file.

Copy Old Source to New Project

We've installed our dependencies and we've told Angular CLI how to find our library assets. But, we still need to bring over the rest of our code. This is just a bunch of copy and paste from the old source directory to the new one:

- Copy **styles/styles.css** to **styles.css**
- Review **com/DotComIt/learnWith/main/main.ts** against **main.ts** in the new project. They were almost identical, but **main.ts** did have some error handling that the original app didn't. I kept the new file as is.
- Copy **com/DotComIt/learnWith/main/app.component.ts** to **src/app/app.component.ts**.
    - The new **app.component.ts** does not have any custom styles or templates or unit tests, so I deleted those custom files from the new project structure.
- Copy **com/DotComIt/learnWith/main/app.module.ts** to **app/app.module.ts**.
- Copy **com/dotComIt/learnWith/model** to **model**.
- Copy **com/DotComIt/learnWith/nav** to **nav**.
- Copy **com/DotComIt/learnWith/services** to **services.**
- Copy **com/DotComIt/learnWith/views** to **views**.
    - Since the location of the views has changed, all the template and CSS sources need to be modified. As an example with the login component. The **@Component**

metadata was like this:

```
@Component({
   selector: 'login',
   templateUrl :
'./com/dotComIt/learnWith/views/login/login.compo
   styleUrls: [
'./com/dotComIt/learnWith/views/login/login.compo
]
})
```

We need to modify the paths to be like this:

```
@Component({
    selector: 'login',
    templateUrl : 'login.component.html',
    styleUrls: [ 'login.component.css' ]
})
```

- Copy **com/DotComIt/learnWith/vo** to **vo.**
- Modify the **index.html**.
    - Change the page title, if necessary.
    - Modify the **app-root** main application variable to this:

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

    - Remove the **favicon.ico** since we don't have one for the new app.
- If you want to retain git version control, then you can copy over your old history from the old application directory to the new project. Just take the **.git** file from the old project and copy it to the new project folder.

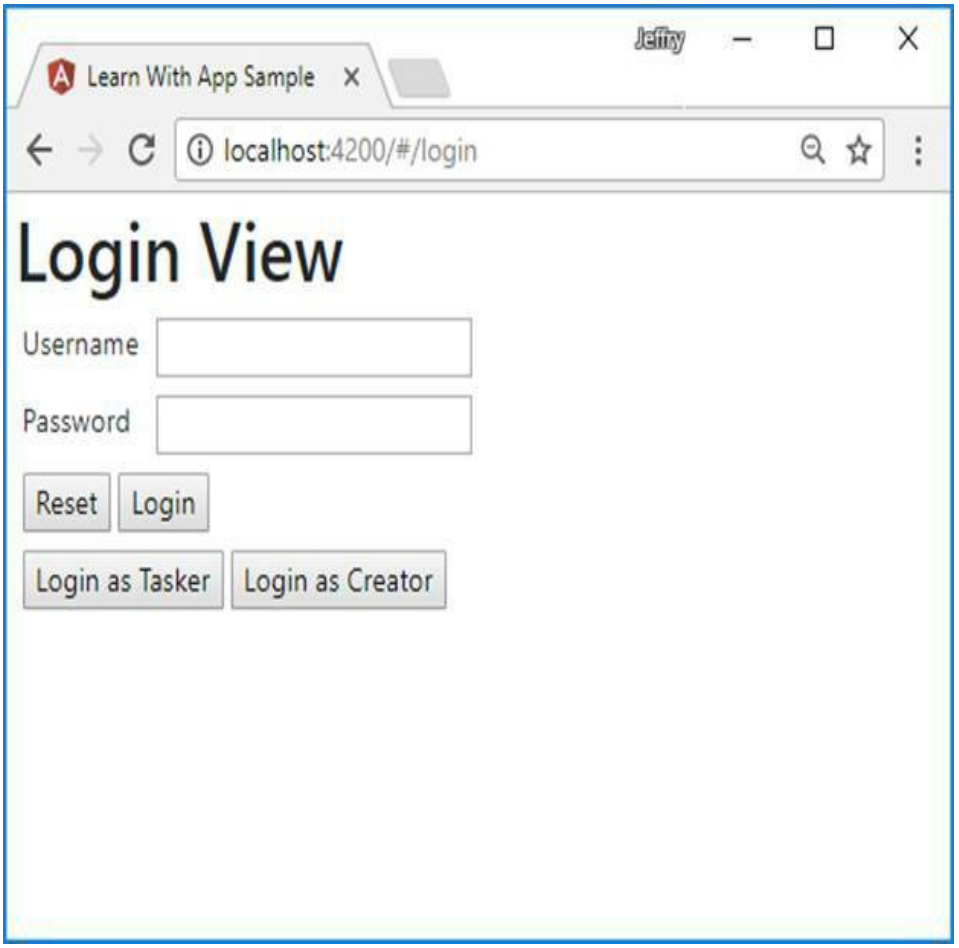
This should cover all the important parts of the new application, migrating it from the old code structure to the Angular CLI structure.

Run the Application

With all code copied, try to run the application:

```
ng serve –open
```

Play around with the app to see what works and see what breaks:



Nothing should break, you should be able to login, view tasks, create tasks, filter tasks, and schedule tasks. Congratulations you converted an application to Angular CLI.

Review Other Features

The purpose of this article was to give you a broad introduction to Angular CLI, but there are things it does that I didn't cover here. Here is a broad list of things:

- **ng lint**: Linting is the process of validating your code without compiling it. It can be used to highlight possible errors that may not be compiler errors.
- **ng test**: This will run your unit tests.
- **ng e2e**: This will start your application and run end to end tests.
- **ng get**: This allows you to retrieve a configuration value. This might be important if you are integrating Angular CLI into other larger build scripts.
- **ng set**: This allows you to set a configuration value. This might be important if you are integrating Angular CLI into other larger build scripts.
- **ng doc**: This accepts a search term and will open the official Angular API documentation for a given keyword. Honestly, I'm more likely to google the keyword to find answers than to use this.
- **ng eject**: This will give you scripts for webpack so that you can build your application without using Angular CLI.
- **Ng xi18n**: This command extracts i18n messages from your templates. This is to help with setting up internalization in your application.

I could write a whole book going into details of this, but I decided to cover the important basics you'll use day to day. If you want to learn more, go to the Angular CLI site.

Chapter 3: Debug Applications from Your Browser

When writing code in any language it is important to understand the tools available to you to help you debug. This article will talk about some of the ways you can debug your code, first by simply outputting data to your web browser console, and second by using a JavaScript step-through debugger.

Use Outputs and Alerts

Sometimes you just want to see the state of a variable at a certain time during your application's run. This allows you to make sure that proper functionality flow is happening and that values are getting properly updated. I used outputs extensively while building the LearnWith Task Manager application.

To give you an example of this, I'm going to turn to the **login.component.ts** file from the LearnWith app. A user tries to login by clicking a login button. The login button executes the **onLogin()** method. Add a line to the method like this:

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

The **console.log()** statement will output some text, "onLogin", to your browser's web developer console. In most browsers you can bring up the web developer console using F12. What this does is show you that your button is properly hooked up to the method. Sometimes this is the first step you need to verify when debugging your code.

Next, the method performs some validation to look for errors. The original implementation of the method would do this:

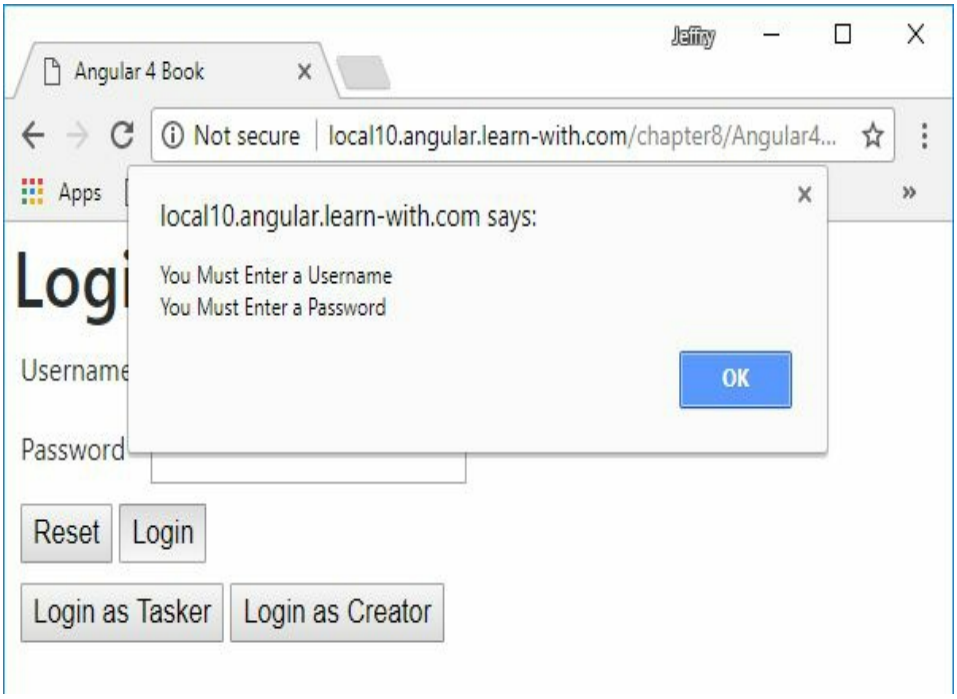
```
if(errorFound == true){
    alert(this.usernameError + '\n' +
this.passwordError);
    return;
}
```

If an error was found, then an alert would be displayed to the user and the processing of the method would stop with the return. Eventually, the alert was replaced with displaying errors directly on the HTML form, which is better user experience. However, the alert was a great first step that could be used to prove that validation was working properly.

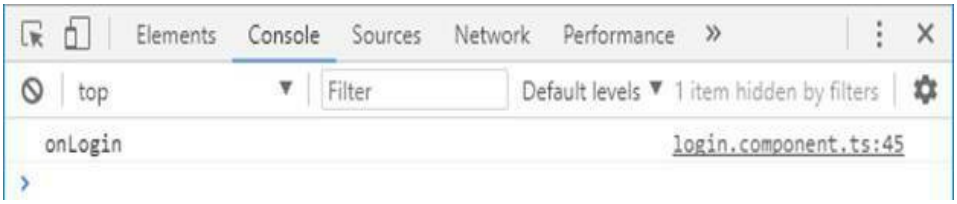
After input validation, the **onLogin()** method is to execute the **authenticate()** method in the **AuthenticationService** object. This method creates a parameter object to pass into the remote service. You can output the object to the console:

```
console.log(parameters);
```

Attempting to login has two console outputs and a single alert. Load the main LearnWith app, and click the login button without entering a username or password. You'll see this:

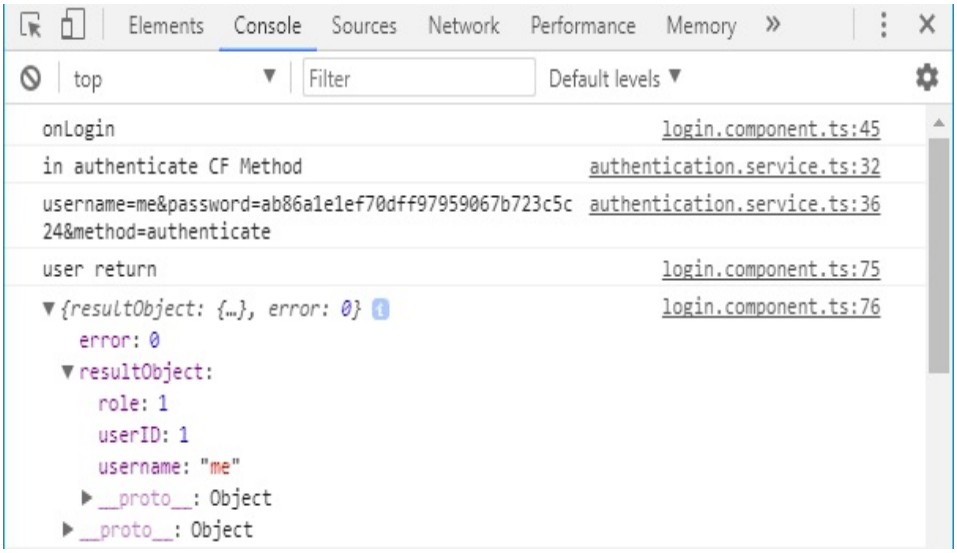


The Chrome developer console will look like this:



The second **console.log()** statement, displaying the parameters object, never ran because there was an error detected, and the return statement prevented the method execution from continuing.

If I enter a username and password, then click login again, this is what the console will look like:

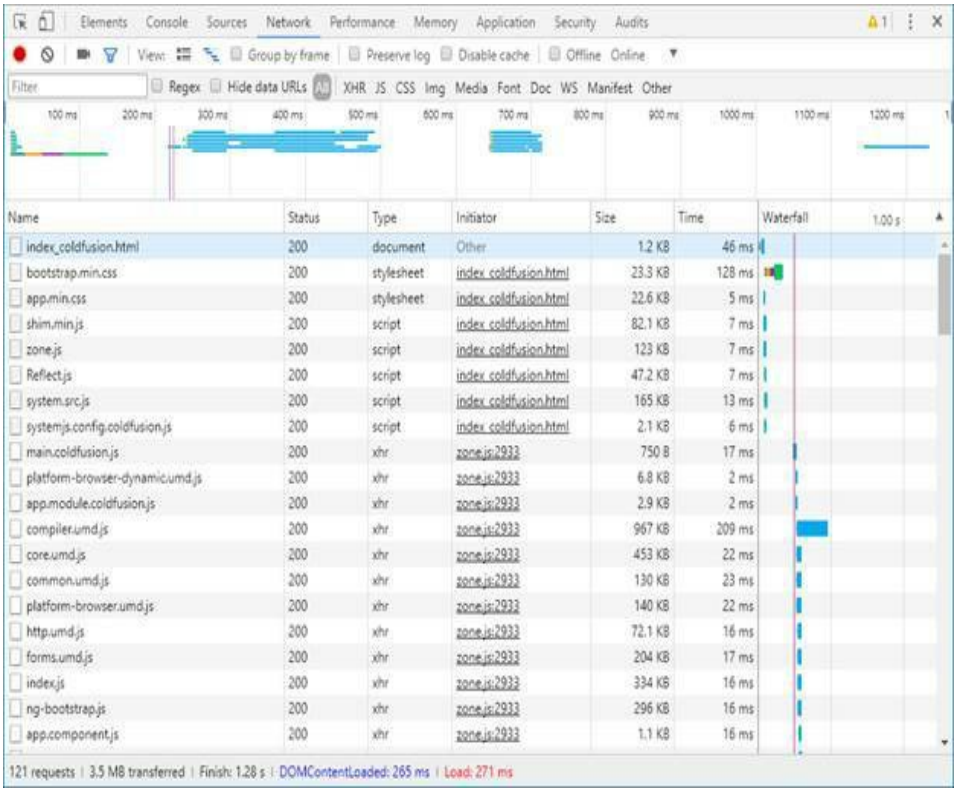


You see the **onLogin()** method, and then the parameter string is output to the console. After that, you see a few log statements from the login success method. The result object returned from the server is output to the screen. You see the arrow next to the object; that means you can drill down into the object's contents to examine what is there. The object returned from the server has two properties: an **error** property and a **resultObject** property. The embedded **resultObject** property is another object, representing the user who logged in. The other two options you see on the screen are both '__proto__'. These relate to JavaScript's Object-Oriented model, which uses prototypes instead of classes, but is beyond the scope of this book.

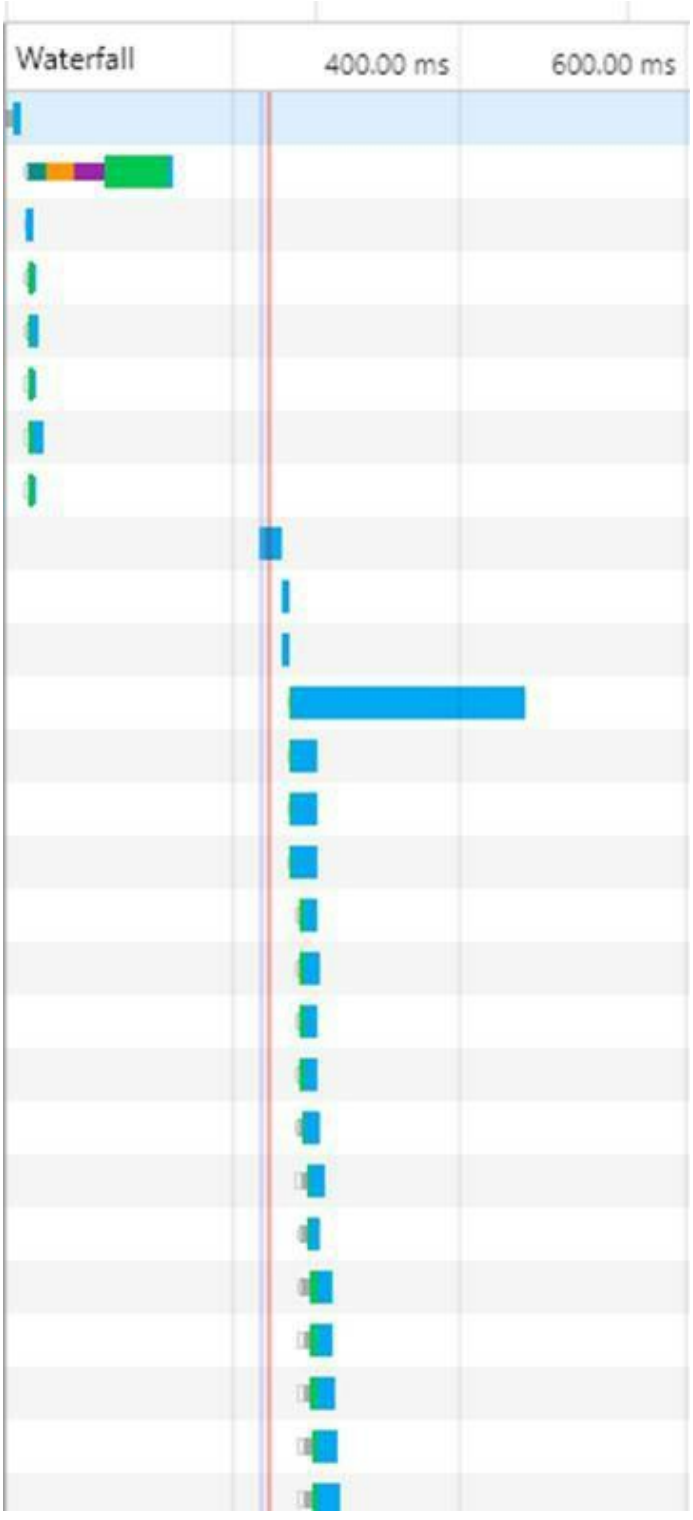
When I started doing JavaScript in the mid-1990s, using alerts was a very common debugging technique. Unfortunately, alert boxes interrupt the application processing flow and make the user dismiss the alert box before they can continue to use the application. This is fine if your intent is to provide the user with feedback that they can't ignore. For debugging information, that is less than ideal. The **console.log()** statements are superior because they do not interrupt the application flow and do not require input from the user to continue. You could also, theoretically, leave them in for deployment without affecting the users of the application.

Review the Network Tab

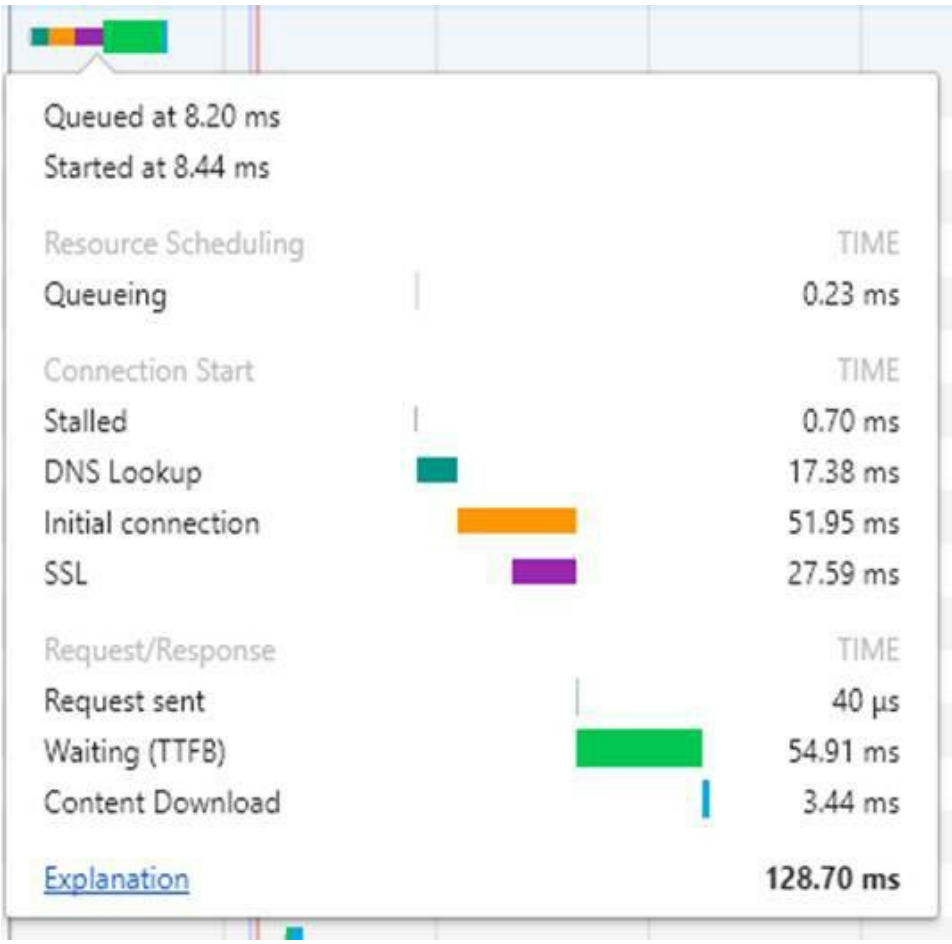
Sometimes you want to review the network calls that pass between your user interface and your server. Many browsers provide this feature built in as part of their web developer tools, and they are all similar. I want to take a deeper look at the Chrome tools which I use daily. This is the Chrome network tab when loading the login page for the LearnWith Task Manager application:



It tells you the file that was loaded; the type of request; the results of the request and what file made the request; and even what line of the file initiated the request. It also gives you the file size and the time it took to load the file. Although not obvious in the above screenshot the Waterfall can give you a good feel for how your page loads:


Waterfall
400.00 ms
600.00 ms

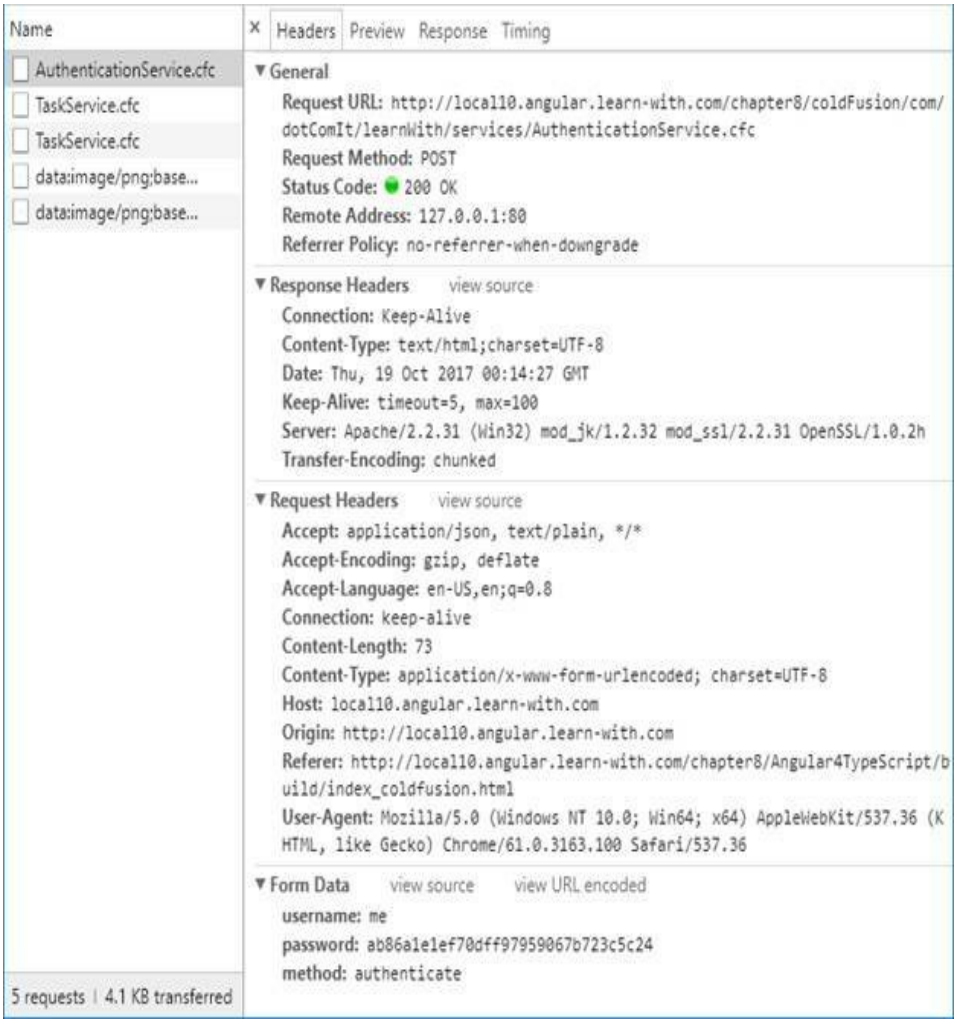
The red line shows where the page was fully loaded. Before the red line you see the length of time it took for the specific file to load. After the red line you'll often see a lot of templates and service calls made by the Angular libraries, since Angular setup does not start until after all the pages and libraries were downloaded. Roll over one of the options to see more details:



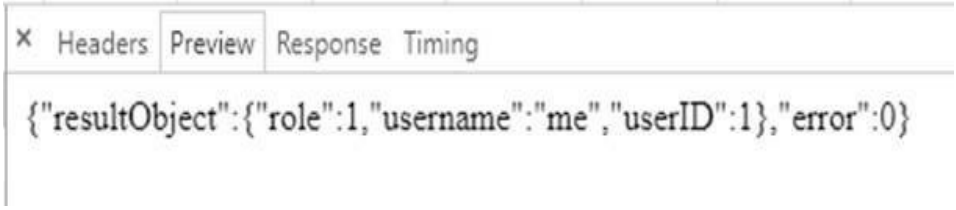
This tells you what time the request was sent; how long it took to download and how long it waited to download. This is interesting information if you are trying to debug network issues that may cause long app load times.

You can use the console to drill into specific details of a request. Just click the file in the network console and the details are brought

up. Here is the request that is used to authenticate the user:

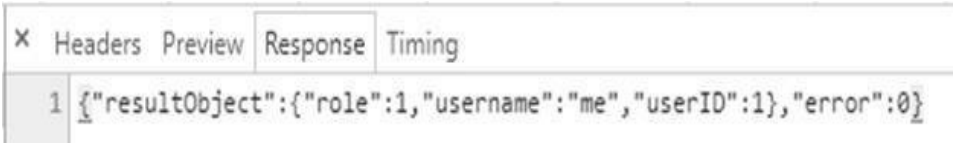


On the right side you see all the details. The form parameters that were sent are in the form data section. You can also see details on the response:



The preview tab shows the results; in this case it gives you a JSON

object you can drill into. The response tab will give you similar data:

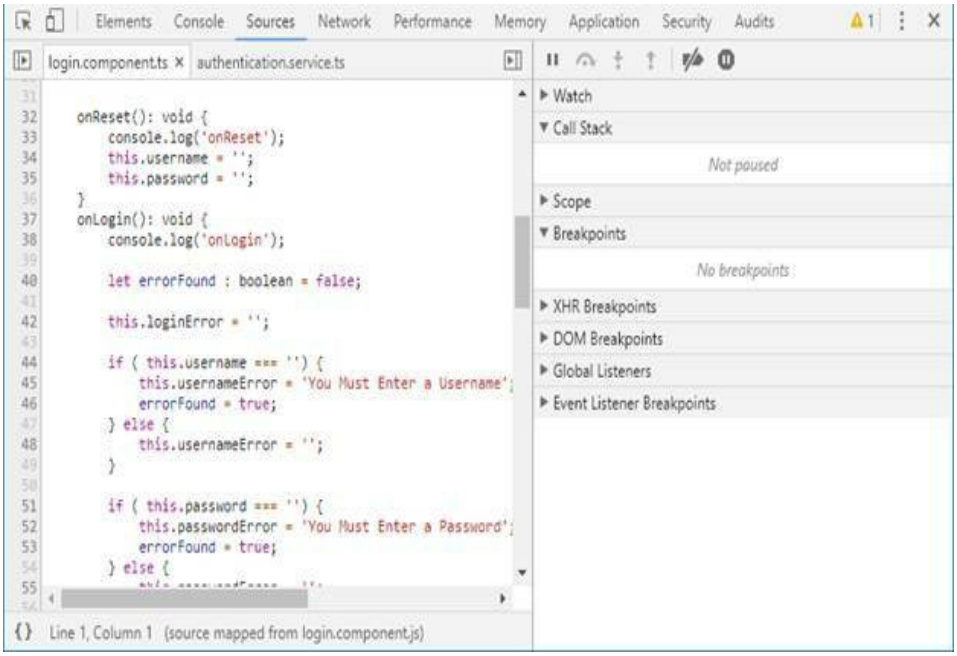


If you are ever having problems with your service calls not working; this is a great way to drill down into the requests to help diagnose the problem. You can also use this tool to find out if your page is loading invalid files, or files from the wrong path. Sometimes one typo in a path name will prevent items from loading and break your whole application.

Use a Step Debugger

I find one of the most useful debugging tools is a step through debugger. Step through debuggers allow you to add breakpoints to code and to manually execute each line so you can review variables and other values. Most browsers have a JavaScript debugger built in to the browser as part of their web developer tools, but for the sake of this article I will focus on the Chrome debugger, because it is the one I use the most.

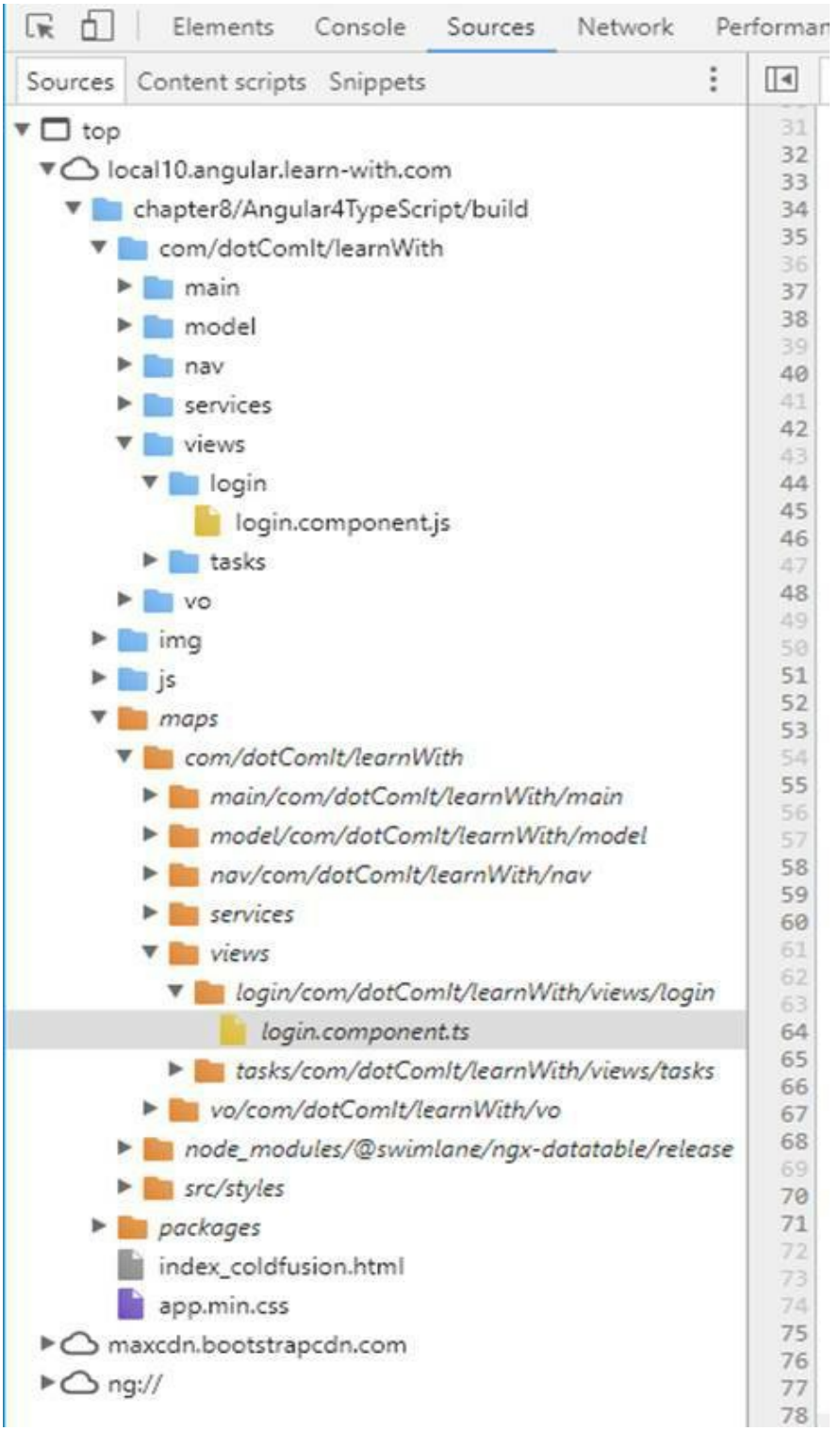
To use the Chrome debugger, first open the web developer tools and then click on the source tab:



The source screen of Chrome developer tools has a tabbed list of files. In the screenshot above, I have two TypeScript files shown, the **login.component.ts** and the **authentication.service.ts**. Even though the browser won't run the TypeScript directly, our use of map files allows us to review the TypeScript code. On the right side of the screen are a bunch of different options, such as watch expressions,

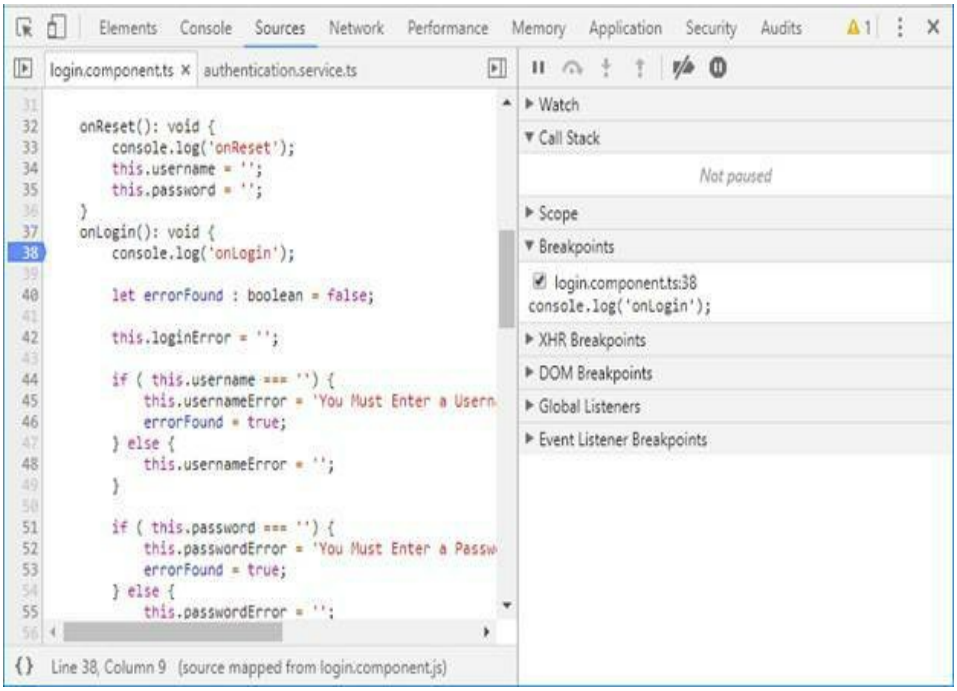
breakpoints and other sections.

To the left of the **login.component.ts** you see a button that kind of looks like a play button from a tape deck. Click it to open up the files view:


Elements
Console
Sources
Network
Performan
Sources
Content scripts
Snippets
top
local10.angular.learn-with.com
chapter8/Angular4TypeScript/build
com/dotComIt/learnWith
main
model
nav
services
views
login
login.component.js
tasks
vo
img
js
maps
com/dotComIt/learnWith
main/com/dotComIt/learnWith/main
model/com/dotComIt/learnWith/model
nav/com/dotComIt/learnWith/nav
services
views
login/com/dotComIt/learnWith/views/login
login.component.ts
tasks/com/dotComIt/learnWith/views/tasks
vo/com/dotComIt/learnWith/vo
node_modules/@swimlane/ngx-datatable/release
src/styles
packages
index_coldfusion.html
app.min.css
maxcdn.bootstrapcdn.com
ng://

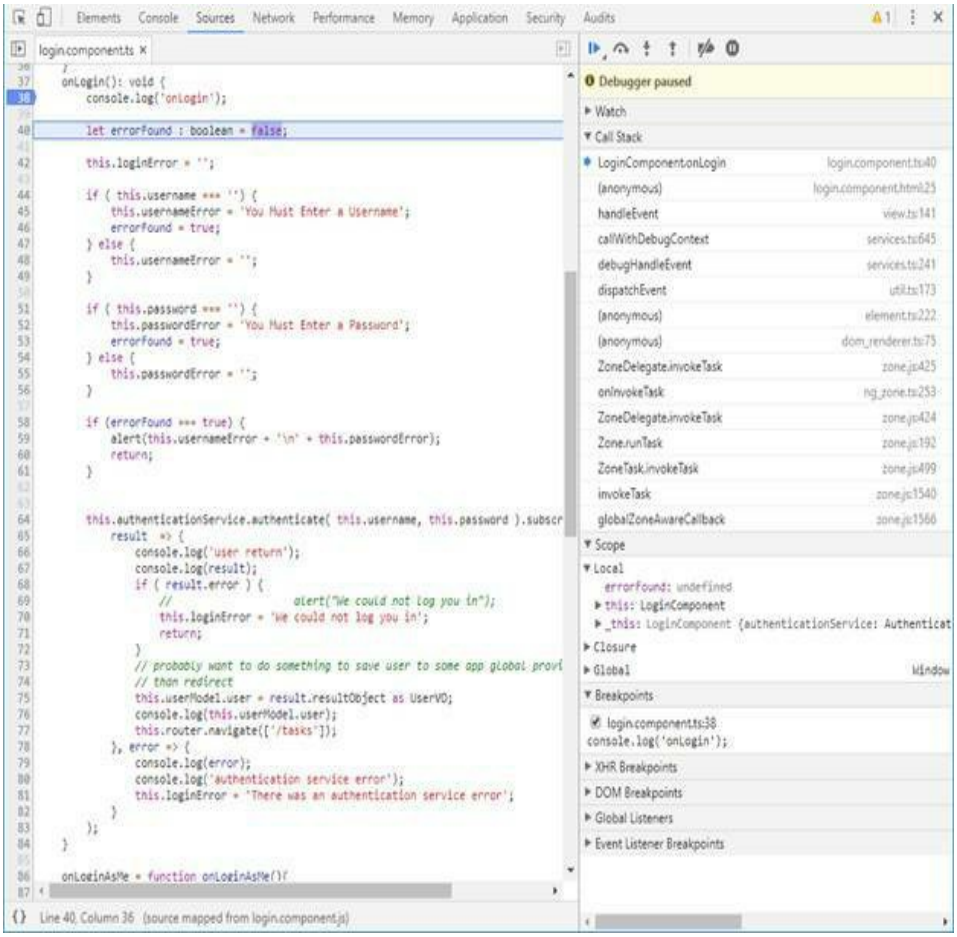
This panel shows you the location of all the files related to the current script. That includes map files, source files, compiled JavaScript files. It even shows some libraries that were loaded. The **ng://** section contains a lot of the Bootstrap components and the HTML templates. You can use this browser to open files not already open. The button on the top right side of the screen can be used to open or close the panel on the right side of the original screenshot.

The main benefit of a debugger is to be able to stop code execution at certain points in code to review the code and step through the code manually. The spot where the code stops is called a breakpoint. You can add a breakpoint by clicking on the line number next to a segment of code:



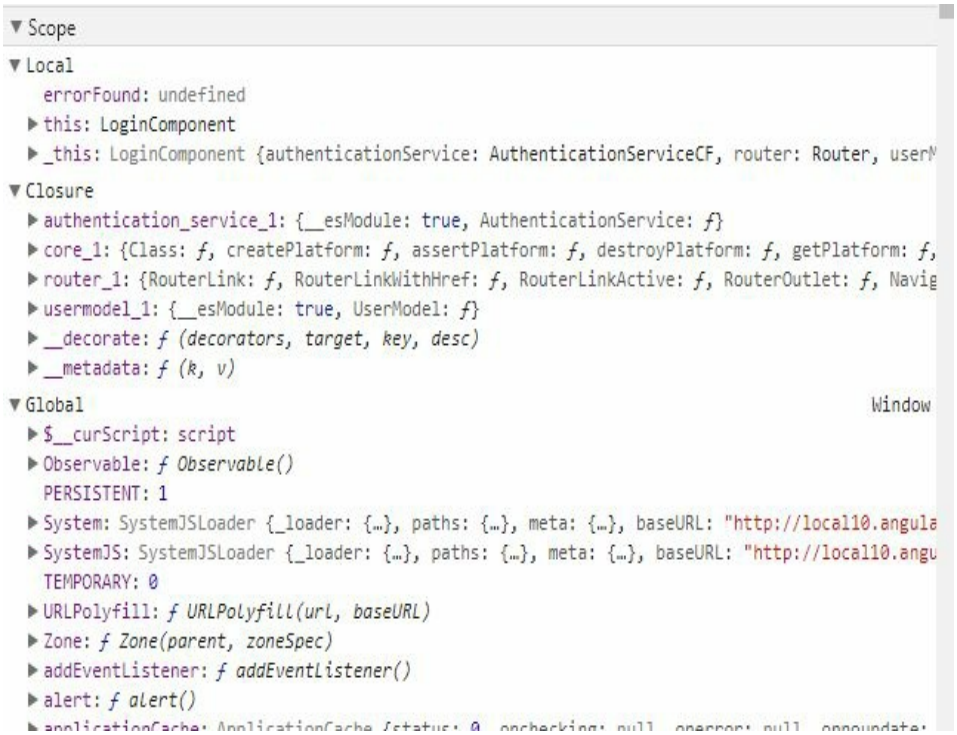
You see in the screenshot that the breakpoint is now highlighted in blue, and the breakpoint shows up in the breakpoints section of the right panel. I put the breakpoint in the first line of the **onLogin()** method. This method is executed when the use clicks the login button of the login screen of the LearnWith application.

Click the login button, and the execution will stop:



The current line that the code is executing is highlighted in the left. On the right you can see the call stack, which contains the series of calls that executed to get to the current line. The elements in the call stack relate to the Angular framework.

You can also see scope variables, which are all the variables that are defined, and accessible from the current method. Here is a more expanded, detailed, version of the scope variables section:



The local variables are defined in the method, and some such as **errorFound** have not been defined yet. The Closure section relates to specific functions available to the method. For example, the **authentication_service_1** is the Angular function that created the **AuthenticationService** object which was injected into the component. The global section refers to a lot of globally accessible variables.

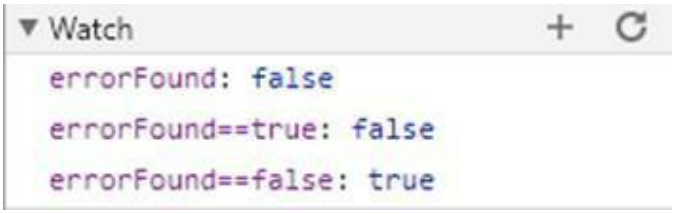
Above the scope, call stack, watch sections you'll see a row of buttons that allow you to control how the debugger works:

Icon	Description
	**Resume**: After execution is stopped at a specific brea button will continue execution for the current line.
	**Step Over**: This button continues onto the next line ir script, ignoring any other function calls

	**Step Into**: This continues onto the next executing line is in another function, then the code will jump into th
	**Step Out**: This button will continue onto the next line code after the current function call completes. Somet this button called “step return” because it immediate results of the function call back to whatever piece of
	**Deactivate Breakpoints**: This button is a helper butt deactivate all breakpoints. I use this most commonly application start up procedure when I don’t want to s debugger until the user is ready to start interacting w application.
	**Pause on Exception**: This button tells the debugger to stop execution when the JavaScript code has an error helpful when you’re trying to diagnose errors, or whe with errors you didn’t even know existed. This button different states; to pause on no exceptions, to pause and to pause on uncaught exceptions.

Most of the buttons in the table should be familiar to you if you’ve used a debugger with other technologies.

Another important aspect of a step through debugger is the ability to set up watch expressions. The watch expression section is in the top right of the window tree. In this window you can specify any expression you wish, and see the results. Here is the watch window with a few watch variables:



I added a simple watch expression, which displays the value of the **errorFound** variable. Then I added two conditions, one for **errorFound== true** and the other for **errorFound == false**. The watch expression will show the results of the expression. Click the plus sign on the right of the header bar to create a new watch expression; then type the expression or variable into the input at the bottom of the watch pane. You can create expressions of any type you need, but mostly I just enter variable names.

Final Thoughts

You should be ready to use these tools to debug your JavaScript application. We discussed using **console.log()** or **alert** to output information to the browser. It showed you how to use the Chrome's network tab to view the requests that the browser sends to the server. Finally, it taught you the concepts behind using Chrome's JavaScript debugger. The debugger is perhaps the most important tool when debugging application. There are even ways to integrate browser debuggers into IDEs, such as IntelliJ; however, that is beyond the scope of this book. I looked at Chrome, but most modern web browsers have similar debugging features for web developers.

Chapter 4: Unit Testing an Angular Application

Angular is designed so that code can easily be tested. In some aspects of programming, such as mission-critical applications or payment processers, unit testing is a key to proving that your changes do not break existing processes.

What is Unit Testing?

Developers have spoken about unit testing and its benefits for years, but if you're like me, you rarely come across clients who are willing to pay you to create unit tests. When they have to make a budget decision between more features or a bulletproof application, "more features" almost always wins. Nonetheless, I thought I'd start with an explanation of what unit testing is and why you might find it useful.

Unit testing is the process of testing how your code will react across many different scenarios. A unit test will run a single discrete chunk of code and verify the results against what you expect. Usually, each test focuses on a single condition, and you write multiple tests against the same set of code. Each test puts the code through different conditions and evaluates the results. Often a unit testing framework will provide a way to run a lot of tests at once and provide you with detailed results.

Let's take the example of the login form from the LearnWith Task Manager. We may want to test these scenarios:

- User clicks login button without entering username or password
- User clicks login button without entering username
- User clicks login button without entering password
- User clicks login button entering both a username and a password

Each one of these scenarios is something that the application must

handle, but all provide different results. If the user does not enter a username, then an error message telling them that they did not enter a username is shown. If the user does not enter a password, then they are shown an error message letting them know to enter a password. If the user does not enter a username or password, then two error messages must be shown. If the user enters both a username and a password, the application should try to call the login service to validate the user. We can create a unit test for each scenario, and the unit test will validate the results.

Setup a Unit Testing Environment

Setting up a unit testing environment for an Angular application took more work than it should be. Both the Angular Quickstart and the Angular CLI projects have testing scripts, but I had lots of problems applying them to the LearnWith code due to versioning issues between third party components like ng-bootstrap and the ngx-datatable. Unit testing draws on a lot of different technologies to work. This section will go into details on the tech stack needed to test an Angular application.

Review the Technology Requirements

Here are some of the technologies we'll use:

- **NodeJS**: NodeJS is server-side JavaScript, and you've used it throughout the book for compiling TypeScript and other actions. If you don't have it installed, go get it.
- **Jasmine**: Jasmine is a unit testing framework and we'll use it to run our code.
- **Karma**: Karma is a JavaScript test runner, created by the AngularJS team. We'll use Karma to execute multiple unit tests.
- **Karma-Chrome-Launcher**: The Karma-Chrome-Launcher is a plugin that allows Karma to launch chrome and load our report results.
- **Karma-Jasmine-HTML-Reporter**: Karma-Jasmine-HTML-Reporter is a Karma plugin that will generate reports directly in a browser and show us the results.
- **Karma-TypeScript**: Karma-TypeScript allows us to write unit tests in TypeScript, transpile them into JavaScript, and then run them through Karma to test.
- **GulpJS**: GulpJS is a script runner that is used for the build process. We can also use it to run tests. The scripts used in the main book are based off GulpJS.
- **@types/jasmine**: The @types/jasmine library contains type definitions that bridge the gap between Jasmine constructs

and TypeScript. They make sure the TypeScript compiler doesn't throw errors.

It is a lot to cover, but I'll detail each step, and once you have a setup, you can just focus on writing tests without worrying about the rest of the infrastructure.

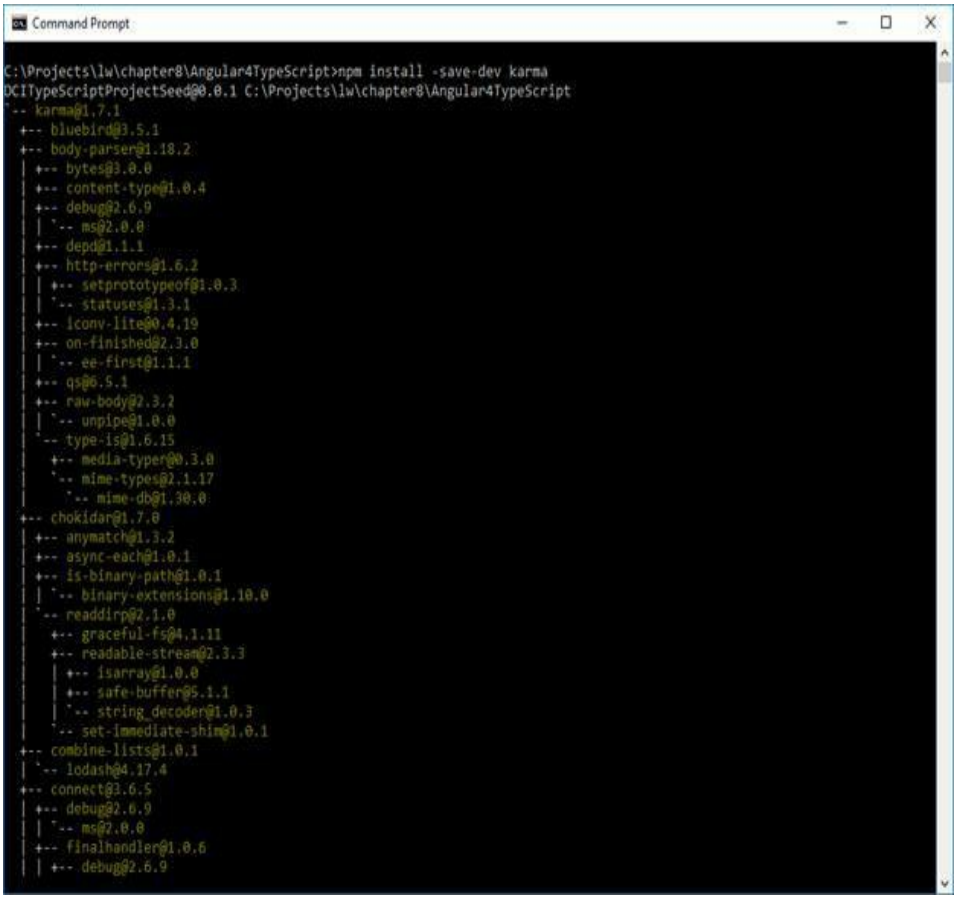
Install Karma, Jasmine, and Utilities

I'm going to assume you already have NodeJS and GulpJS installed. Let's focus on the new things.

First, install Karma so we can run unit tests. Execute this command:

```
npm install –save-dev karma
```

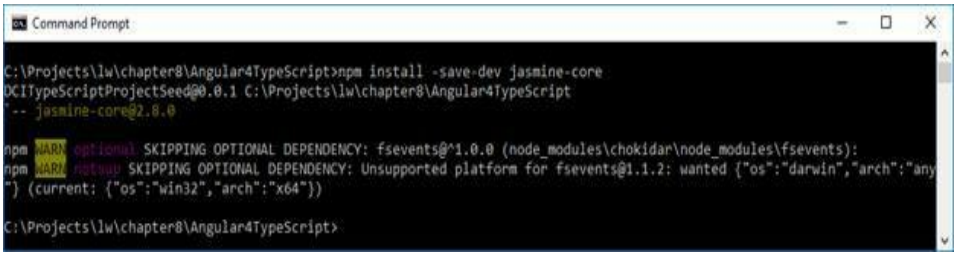
You'll see something like this:



Next, install Jasmine, the unit testing framework:

```
npm install –save-dev jasmine-core
```

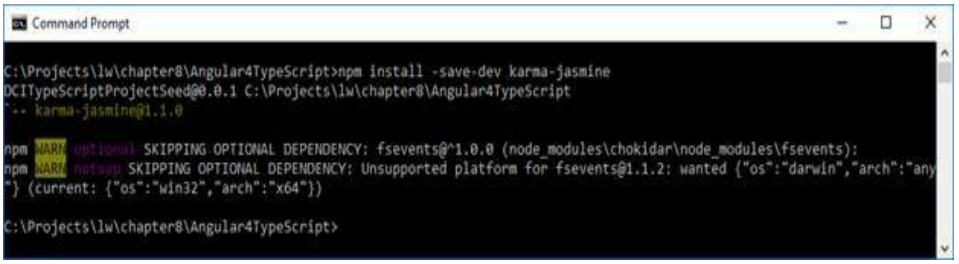
We only need to install the jasmine-core module. You'll see something like this:



Now install the [Karma plugin for Jasmine](), so Jasmine and Karma can work together easily:

```
npm install –save-dev karma-jasmine
```

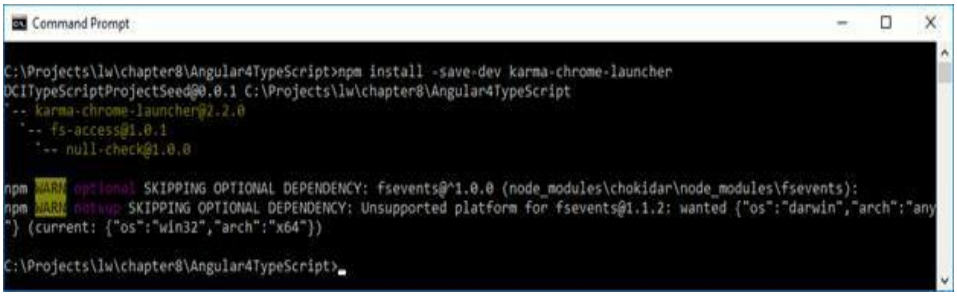
You should see this:



Now we need the Karma Chrome Launcher, which will allow our unit tests to show results in the chrome browser:

```
npm install –save-dev karma-chrome-launcher
```

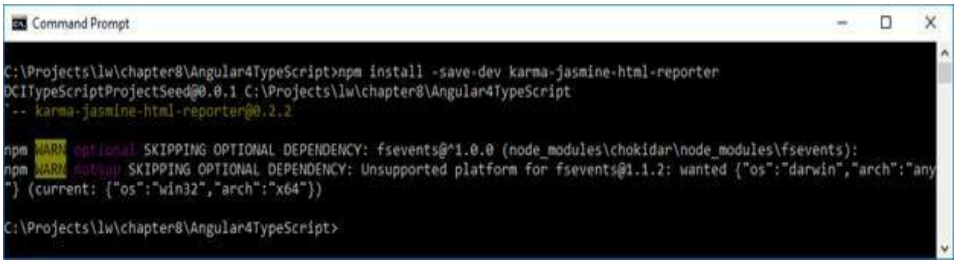
I assume you already have Chrome installed because if you're reading this you're a web developer already. You'll see this roll by the console:



Then install the HTML reporter which is used to display results inside the browser:

```
npm install -save-dev karma-jasmine-html-reporter
```

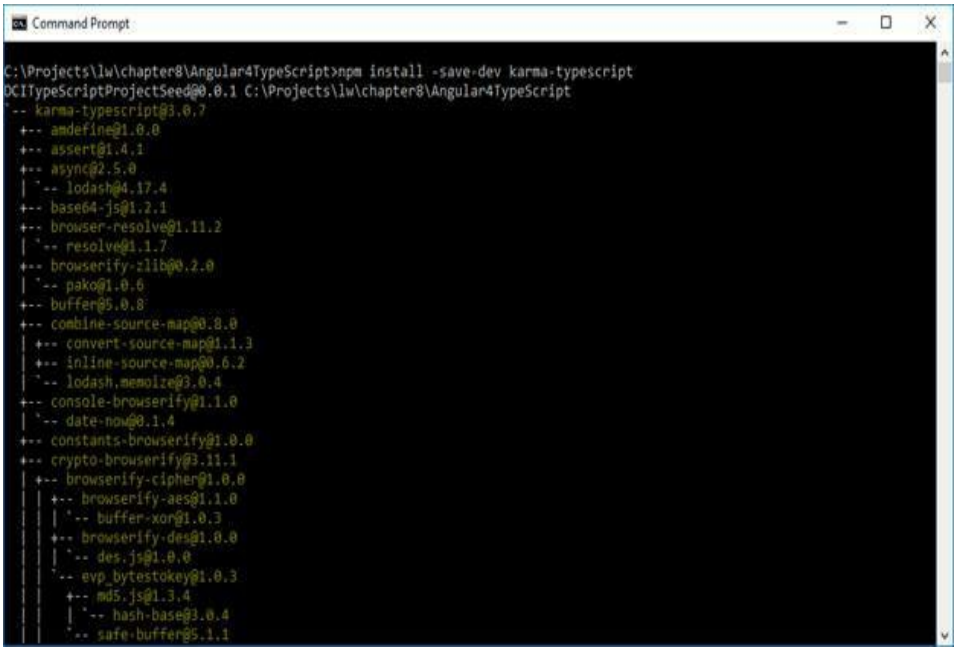
You'll see something like this:



Now install the karma-typescript plugin:

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

This allows us to write our tests in TypeScript and compile them automatically by Karma. A lot of stuff will flash by your console:



Finally, install the jasmine types so that our TypeScript compiler will recognize Jasmine constructs in code:

```
npm install –save-dev @types/jasmine
```

That is it for your new NodeJS packages.

Create Karma Configuration

Create a file named **Karma.conf.js**. Add the start of the configuration:

```
module.exports = function(config) {
}
```

Karma will know to look for **module.exports** and run the related function, which will define the configuration for unit tests. Inside the function, call the **set()** function on the **config** object:

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

The set function accepts a single object as the argument. It is blank in the code segment above, so start filling it in. First, specify the browser:

```
browsers: ["Chrome"],
```

Our unit tests will be run using Chrome. Then specify the frameworks:

```
frameworks: ["jasmine", "karma-typescript"],
```

We're using Jasmine as our unit testing framework and karma-typescript to compile our unit tests from TypeScript to JavaScript. Now, configure karma-typescript:

```
karmaTypescriptConfig: {
    tsconfig: "./tsconfig.test.json",
},
```

The **tsconfig.test.json** file is a configuration file for the TypeScript compiler. A file, **tsconfig.json** already exists in your project, but we want to use a slightly different file for testing purposes. We'll review that after creating the **karma.conf.js**.

Next, we need to specify a minetype for the TypeScript files:

```
mime: {
    'text/x-typescript': ['ts', 'tsx']
},
```

Without this line, the browser will mistake the TypeScript files for video files and the test will just error out.

Now, tell karma-typescript to compile files:

```
preprocessors: {
    "**/*.ts": ["karma-typescript"],
},
```

The first element is a glob which tells karma-typescript to process all files with a **ts** extension.

Finally, specify our reporters:

```
reporters: ["progress", "kjhtml"]
```

The reporters are how Karma will show us results. We specified

two, “progress” which will show up in the console and “kjhtml” which refers to the karma-jasmine-html-reporter, which will show results in the browser. A cool thing about using the browser to show results is that we can use the browser’s debugging tools debug broken tests. It can be useful at times.

That completes the **karma.conf.js** configuration file. Normally we’d specify files to include, or exclude, however I opted to make that stuff more dynamic in our Gulp script.

Before moving onto the Gulp script additions, let’s look at the **tsconfig.test.json** file:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "lib": [ "es2015", "dom" ],
    "noImplicitAny": true,
    "suppressImplicitAnyIndexErrors": true,
    "outDir": "compiledtests"
  },
  "types": ["jasmine"],
  "exclude": [
    "node_modules"
  ]
}
```

This file is almost identical to the **tsconfig.json** file, so I won’t go into it in details. It has one important addition I want to highlight:

```
"types": ["jasmine"],
```

This line tells the compiler to load the jasmine type files from the @types/jasmine library. This allows karma-typescript to compile the Jasmine constructs inside of TypeScript without error.

Run through GulpJS

Since we are using GulpJS for most of the build tasks around this app, I want to run unit tests through GulpJS too. This is pretty easy to do, as we have already installed everything we need. Open the **gulpfile.js**, and load the Karma library:

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

Now create a test task:

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

Create a new karma server in the task:

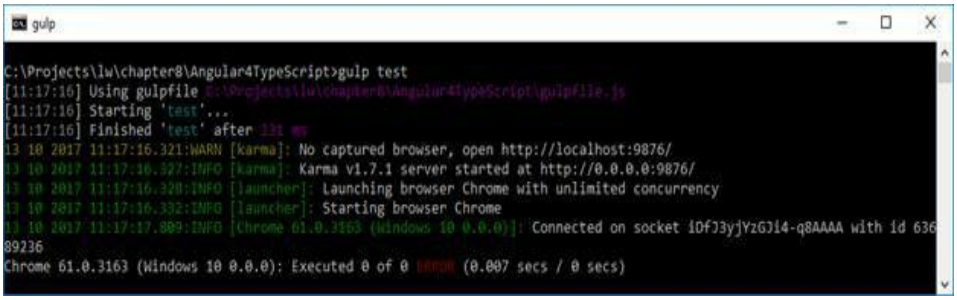
```
new karma.Server({
    configFile: __dirname + '/karma.conf.js',
}).start(gulp);
```

This uses the **karma** object and references the **Server()** method on it. The **Server()** method accepts a single argument that represents a configuration object. We are telling it to use the config file we previously created. The **__dirname** variable is a special NodeJS variable representing the directory to the executing file. After the **Server()** is created, the **start()** is called on it; this executes the unit tests.

You can now run the **Karma** tests through Gulp, like this:

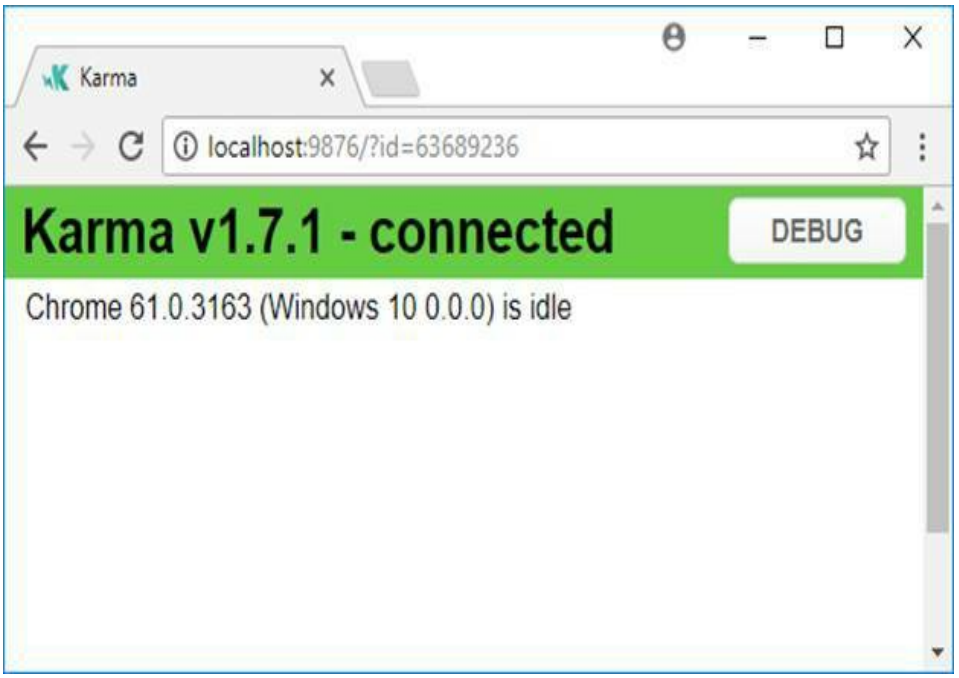
```
gulp test
```

You'll see results similar to this:



It didn't find any tests to run, yet which is expected. Chrome should

also have been launched:



An interesting thing about the Gulp execution is that we can use it to modify the Karma configuration on the fly. We're going to do that to tell Karma what files to process, and which files to avoid.

Let's focus on the files we want to include. These will be all the source files, all the test files, and all the Angular component templates and CSS files. First, open the **config.js** file. Find the **baseDirs** object and add a **testRoot** property:

```
testRoot : "tests/"
```

Now find the **configObject** variable. First, we want to add a value to look at all the tests:

```
typeScriptTestSource : [baseDirs.testRoot +
"**/*.ts"],
```

We also want a value for all the HTML templates in the com directory:

```
htmlTemplateSource : [baseDirs.sourceRoot +
```

```
baseDirs.codeRoot +
                          '/**/*.html'],
```

This is similar to the **htmlSource** value that already exists, but this one just pulls templates from the **codeRoot** directory--**com**—and not from the **root** directory. We'll need to tell Karma to process the app's TypeScript files, but a value for that already exists and is named **typeScriptSource**. We'll need to tell Karma to process the CSS files, but for that we can use the existing **cssStyleURLsSource** value.

Go down to the **staticConfig** object and put this all together:

```
testFilePatterns : [
  { pattern: baseDirs.testRoot + "base.test.ts"
},  // new file
  { pattern: configObject.typeScriptSource[0]
},    // existed
  { pattern: configObject.htmlTemplateSource[0]
},  // new
  { pattern: configObject.cssStyleURLsSource[0]
},  // existed
  { pattern: configObject.typeScriptTestSource[0]
} // new
]
```

We are creating patterns that will tell Karma how to find files. First, it processes the **base.test.ts** file. We'll use it place for a lot of shared code used by our unit tests. Create an empty file at **tests/base.test.ts** so we don't forget later.

Then the **typeScriptSource** is added so that our application code is processed. The **htmlTemplateSource** is added, Karma is aware of all our HTML templates. Then the CSS is added with the **cssStyleURLsSource** value, so Karma knows to find the CSS loaded on the fly by Angular. Finally, our tests are added using **typeScriptTestSource**.

Go back to the **gulpfile.js** and find the test task. Add the files to the **karma.Server()** config object:

```
new karma.Server({
    configFile: __dirname + "/karma.conf.js",
    files : config.testFilePatterns,
}).start();
```

Moving right along. Next, we want to set up a proxy. The server built into Karma serves up all our code from an internal directory, **base**. All our templates are loaded under the assumption that our code will be served from the root. We need to tell Karma to automatically redirect all our template loads from the root directory, **/**, to the base directory, **/base**.

To keep things flexible, I'm going to go back to **config.js** and add a variable for the test web root:

```
testWebRoot : "base/"
```

I added this to the **baseDirs** object. Now switch back to the **gulpFile.js** and add a proxies entry to the **karma.Server()** config**:**

```
proxies: {
    "/com/" : "/" + config.testWebRoot +
config.sourceRoot + config.codeRoot
},
```

When the server is running, this will redirect all calls from "**/com**" to "**/base/src/com**".

Configure Tasks for Different Services Integration

Finally, we want to tell **Karma** to ignore the ColdFusion and NodeJS specific files, including the services integration and the main module definition. We are dealing with three different use cases here. One for testing an application that loads our mock services, one for testing an application that loads the ColdFusion services, and one for testing an application that loads the NodeJS services.

Services	Mock	ColdFusion	N
Mock	*Include All*	Exclude Modules, Tests, and Services	Ex Te
	Include Mock Services		Ex

ColdFusion	*Exclude Module and Tests*	*Include All*	Te
NodeJS	*Include Mock Services* *Exclude Module and Tests*	Exclude Modules, Tests, and Services	*In*

The mock services are included for all test suites, so they are loaded in our default module definition, and then overwritten for ColdFusion or NodeJS specific tests. The problem of having to exclude certain directories is so that similarly named services do not get confused with each other. It is a problem created purposely by the multi-use of shared code between the other books in this series.

Open the **config.js** file and create a new object named **testingDirsToIgnoreObject**:.

```
var testingDirsToIgnoreObject = {
}
```

First thing we want to add will be the main application:

```
mainApp : baseDirs.sourceRoot + baseDirs.codeRoot
+
          '/dotComIt/learnWith/main/main.*.ts',
```

We want to avoid compiling this into the testing code because it's only purpose is to load the **AppComponent**, which we will be replacing with a mock application module. The **mainApp** value reuses the **baseDirs.sourceRoot** and **baseDirs.codeRoot**, because we love encapsulation.

Now create three variables for the default Mock specific code. Start with the **AppComponent**:

```
defaultModule : baseDirs.sourceRoot +
baseDirs.codeRoot +

'/dotComIt/learnWith/main/app.module.*.ts',
```

As previously stated, we'll be replacing the **app.module** with a mock application. Create a variable that points to the mock services:

```
defaultServicesSource : baseDirs.sourceRoot +
baseDirs.codeRoot +

'/dotComIt/learnWith/services/mock/**/*.ts',
```

Create a value that points to the Mock Services tests:

```
defaultTestSource : baseDirs.testRoot +
baseDirs.codeRoot +

'/dotComIt/learnWith/services/mock/**/*.ts',
```

The default module and test source will be ignored when running NodeJS or ColdFusion specific unit tests, but these lines will be included as part of the default tests, but removed elsewhere.

Let's set up similar variables that point to ColdFusion specific files. First the ColdFusion version of the **AppComponent**:

```
CFModule : baseDirs.sourceRoot +
baseDirs.codeRoot +

'/dotComIt/learnWith/main/app.module.coldfusion.t
```

Now, create a variable that points to the CF Specific tests:

```
CFTestSource : baseDirs.testRoot +
baseDirs.codeRoot +

'/dotComIt/learnWith/services/coldfusion/**/*.ts'
```

We obviously haven't created these yet, but you'll notice that the test code structure will parallel the application code structure. Finally, add a variable for the ColdFusion services source code:

```
CFServicesSource : baseDirs.sourceRoot +
baseDirs.codeRoot +

'/dotComIt/learnWith/services/coldfusion/**/*.ts'
```

We'll mirror the same three variables for NodeJS:

```
NodeJSModule : baseDirs.sourceRoot +
baseDirs.codeRoot +

'/dotComIt/learnWith/main/app.module.nodejs.ts'
NodeJSServicesSource : baseDirs.sourceRoot +
baseDirs.codeRoot +

'/dotComIt/learnWith/services/nodeJS/**/*.ts',
NodeJSTestSource : baseDirs.testRoot +
baseDirs.codeRoot +

   '/dotComIt/learnWith/services/nodeJS/**/*.ts',
```

Now, we can combine these 10 values to create the exclude arrays for each set of unit tests well want to run. For the mock services:

```
defaultDirsToExclude : [
    testingDirsToIgnoreObject.mainApp,
    testingDirsToIgnoreObject.CFModule,
    testingDirsToIgnoreObject.CFServicesSource,
    testingDirsToIgnoreObject.CFTestSource,
    testingDirsToIgnoreObject.NodeJSModule,

testingDirsToIgnoreObject.NodeJSServicesSource,
    testingDirsToIgnoreObject.NodeJSTestSource
],
```

For the ColdFusion services:

```
CFDirsToExclude : [
    testingDirsToIgnoreObject.mainApp,
    testingDirsToIgnoreObject.defaultModule,
    testingDirsToIgnoreObject.defaultTestSource,
    testingDirsToIgnoreObject.NodeJSModule,

testingDirsToIgnoreObject.NodeJSServicesSource,
    testingDirsToIgnoreObject.NodeJSTestSource
],
```

And for the NodeJS Services:

```
NodeJSDirsToExclude : [
    testingDirsToIgnoreObject.mainApp,
    testingDirsToIgnoreObject.defaultModule,
    testingDirsToIgnoreObject.defaultTestSource,
    testingDirsToIgnoreObject.CFModule,
    testingDirsToIgnoreObject.CFServicesSource,
    testingDirsToIgnoreObject.CFTestSource,
],
```

I put each of these variables in the **staticConfig** object of the **config.js** file.

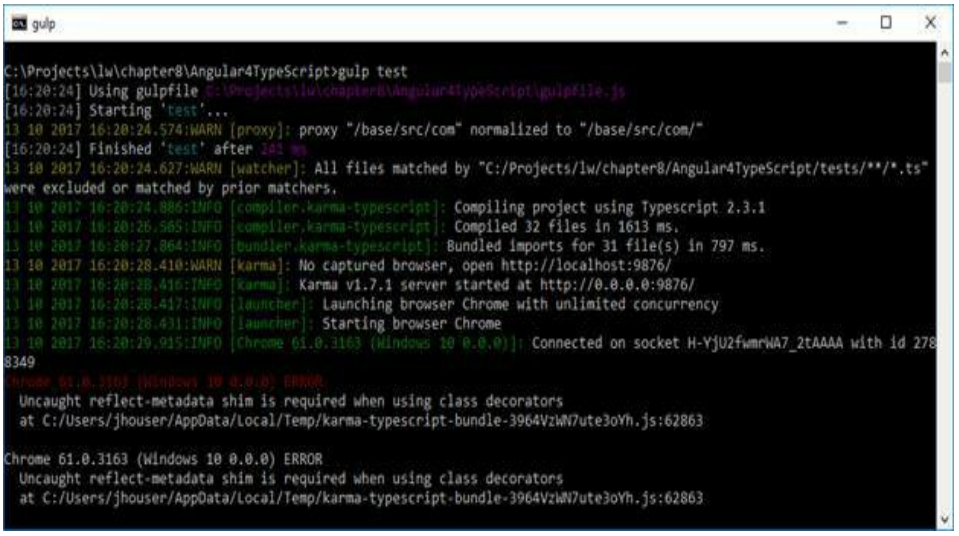
Now loop back to the **gulfile.js**. Add the exclude property to the test. Here is the final task:

```
gulp.task("test", function () {
 new karma.Server({
    configFile: __dirname + "/karma.conf.js",
    files : config.testFilePatterns,
    proxies: {
     "/com/" : "/" + config.testWebRoot +
config.sourceRoot + config.codeRoot
    },
    exclude : config.defaultDirsToExclude,
 }).start();
});
```

We should be able to run the test task, now, right? Give it a shot:

```
gulp test
```

You'll see something like this:



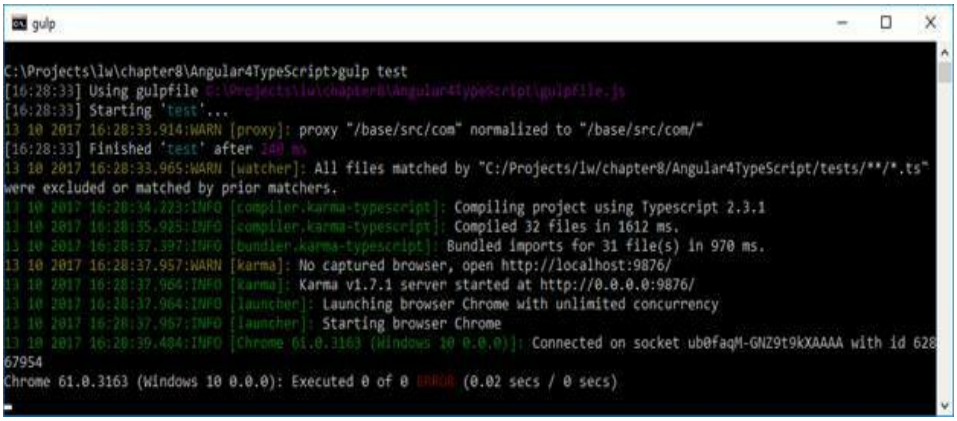
Wait? That's an error and is not the ideal result we're after. What happened? Well, when we run the app in the browser, we are manually including a bunch of libraries that Angular relies upon. From our index file:

```
<script src="js/core-js/client/shim.min.js">
</script>
<script src="js/zone.js/dist/zone.js"></script>
```

These libraries aren't being loaded yet. Open the **base.test.ts** file in the tests directory. Add these lines:

```
import "core-js"
import "zone.js/dist/zone";
import "zone.js/dist/long-stack-trace-zone";
import "zone.js/dist/proxy";
import "zone.js/dist/sync-test";
import "zone.js/dist/jasmine-patch";
import "zone.js/dist/async-test";
import "zone.js/dist/fake-async-test";
```

Now try to rerun the tests:



We don't have any tests yet, but at least aren't running into errors either.

Create the Testing Tasks for Alternate Services

We can make easy modifications to the Gulp task to include the ColdFusion services or the NodeJS services. We created the different exclude variables, so we don't include multiple services named identically, as those will just confuse the unit testing scripts.

To test the ColdFusion services or NodeJS services, you can just modify the exclude directory. This is the ColdFusion task:

```
gulp.task("testColdFusion", function () {
    new karma.Server({
        configFile: __dirname + "/karma.conf.js",
        files : config.testFilePatterns,
        proxies: {
          "/com/" : "/" + config.testWebRoot +
config.sourceRoot +
                    config.codeRoot
        },
        exclude : config.CFDirsToExclude,
    }).start();
});
```

And here is the NodeJS test task:

```
gulp.task("testNodeJS", function () {
    new karma.Server({
```

```
        configFile: __dirname + "/karma.conf.js",
        files : config.testFilePatterns,
        proxies: {
            "/com/" : "/" + config.testWebRoot +
config.sourceRoot +
                      config.codeRoot
        },
        exclude : config.NodeJSDirsToExclude,
    }).start();
});
```

All test runners will show similar results at this point. Now it is time to start creating tests.

The Simplest Test

I'm going to start by creating a simple proof-of-principle test, but I'll expand onto more details. Create a file named **sample.test.ts** in the **tests/com/dotComIt/learnWith/main** directory. This mirrors the directory structure of our source code in the Angular app.

I'll take a moment to cover a few diversions. First, I named the file with a '.test.ts' extension to signify that it is a test. This naming convention comes from past experience, however it is common in the Angular world to use the '.spec' post fix instead of '.test'. If you use that naming style, the name would be **sample.spec.ts**. I prefer to use test, as I find it more descriptive.

The second diversion is the directory structure. I've organized the directories so that the tests are completely independent of the actual code. Many people prefer to put the tests and the code in the same location. I'm not sure which is better, but both seem suitable.

Now, let's create the test. First, add code like this:

```
describe('Sample Tests', function(){
});
```

The **describe()** function is a Jasmine function that defines a test suite, which is a collection of related tests. The function takes two arguments. The first argument is a string that is the name of the test suite, in this case 'Sample Tests'. The second argument is a function, which runs the tests. You can embed a **describe()** function inside other **describe()** functions if you feel it is relevant to your testing.

Inside the **describe()** function, create your first test:

```
it('True is True', function(){
    expect(true).toBe(true);
});
```

The **it()** function defines the test. It has two arguments, and the first is the descriptive name of the test. If it fails, you can use this name

to determine which test actually failed. The second argument is a function that performs the test.

Inside the **it()** function is another function call named **expect()**. I won't get into its internals, but the naming conventions are designed to be easily readable. This line:

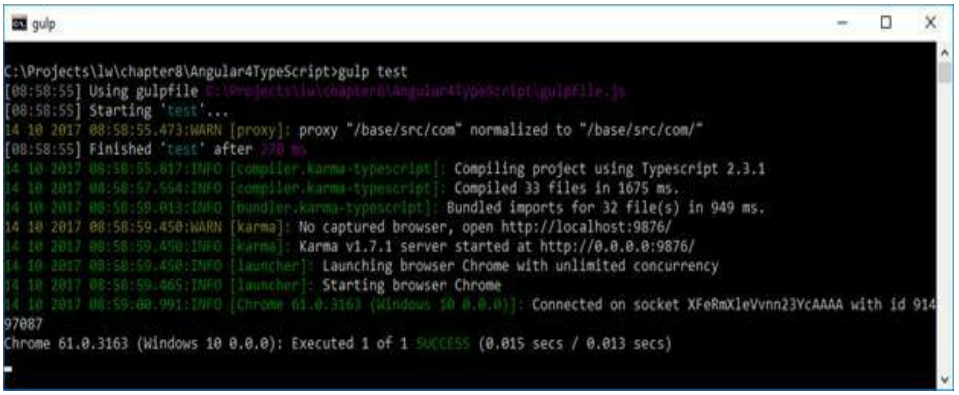
```
expect(true).toBe(true);
```

says that "I expect true to be true." You've seen the 'dot' syntax used to chain functions in the Angular code.

Run the test suite:

```
gulp test
```

You'll see this:

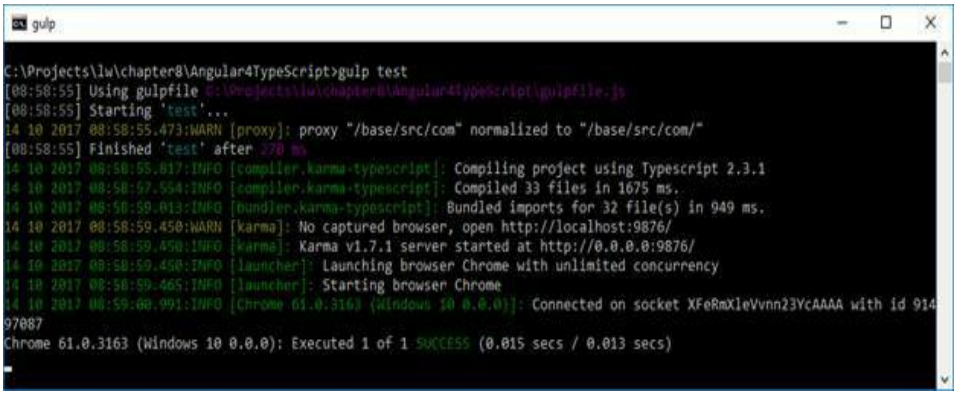


One test successfully executed. Congratulations. Notice that you are not put back to a command prompt. The test runner will stay open, review code, and rerun if you make changes.

We only put one check in the test, but let's add another:

```
expect(false).toBe(false);
```

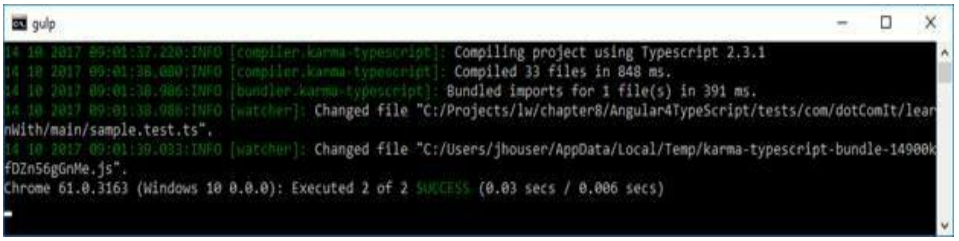
Rerun the test:



You see the same results. Even though there are two checks, there is only one test. However, it doesn't make sense to check if false is false in a test named "True is True". Let's break that out into an independent test:

```
it('False is False', function(){
    expect(false).toBe(false);
});
```

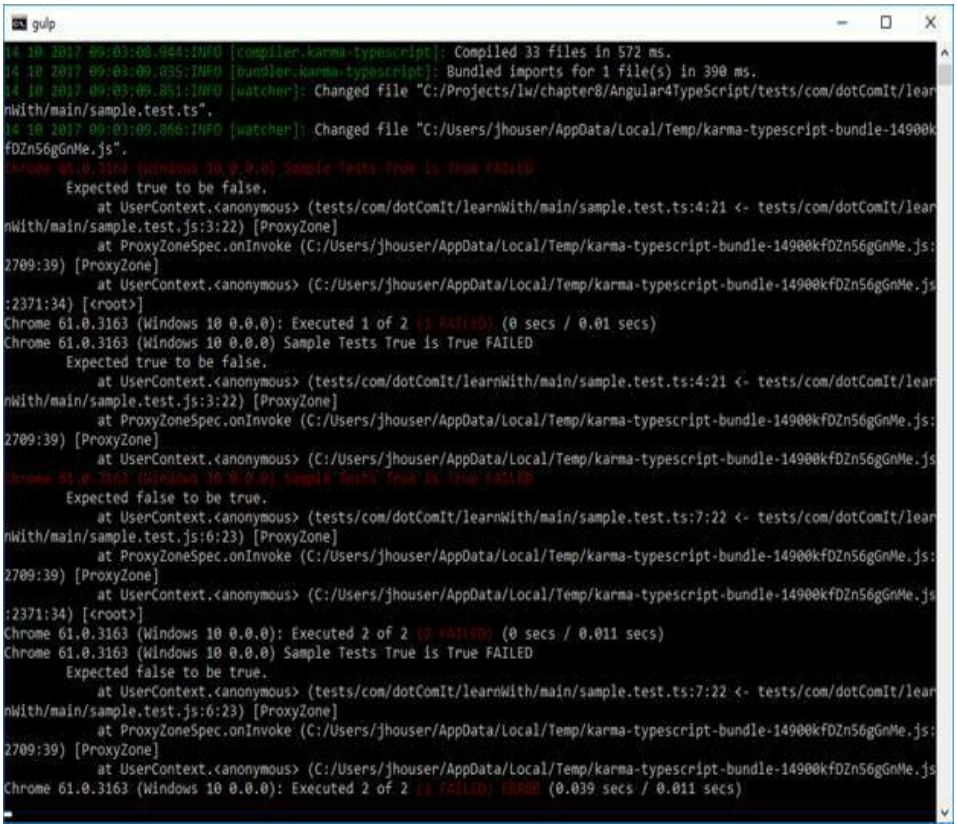
If you kept the test runner up, you'll see these results:



If not, you can run it again to see similar changes. Everything is going well. Let's force a failure by changing both tests:

```
it('True is True', function(){
    expect(true).toBe(false);
});
it('False is False', function(){
    expect(false).toBe(true);
});
```

True should not be false, and false should not be true, so both tests should fail. Rerun the tests:



In the error message, it tells us which tests failed—in the red. It also gives us a stack trace, so we have the line item that caused the error.

This is, in general, how unit testing works. You run some code and validate the actual results against your expected results. Now we can move on to start something real.

Test Classes without Angular Dependencies

Despite building an Angular application, we did create some classes that do not have any Angular dependencies: the value objects and model classes. Value objects do not need to be tested since they are just data containers and contain no functionality. However, the model classes do contain some functions, and those should be tested. I'm going to focus on the **UserModel**, however the code repository will also contain tests for the **TaskModel**.

Review the UserModel

This is the code for the **UserModel**:

```
import {UserVO} from "../vo/UserVO";
import {Injectable} from "@angular/core";
@Injectable()
export class UserModel {
    user : UserVO;
    readonly TASKER_ROLE = 1;
    readonly CREATOR_ROLE = 2;
    validateUser() : boolean {
        if ( !this.user) {
            return false;
        }
        return true;
    }
    isUserInRole(roleToCompare:number):boolean {
        if (!this.user) {
            return false;
        }
        if (this.user.role === roleToCompare) {
            return true;
        }
        return false;
    }
};
```

I know I said this doesn't have any Angular dependencies, but that is

not completely true. It does import the **Injectable** metadata so that Angular knows it can use this class as something that can be injected into a component. However, that is just a message to the Angular framework and does not prevent us from using this class outside of it.

The class has one instance variable, an instance of the **UserVO**. It has two functions:

- **validateUser()**: Used to determine if the user has logged in yet or not.
- **isUserInRole()**: Used to determine if the user is in a role. The UI code uses this to determine if the user has proper permissions to access something.

We will write tests around these two functions.

Write the Tests

Create a file named **usermodel.test.ts** in the **tests/com/dotComIt/learnWith/**model directory. As always start with some imports:

```
import {UserModel} from

"../../../../../src/com/dotComIt/learnWith/model/
import {UserVO} from
"../../../../../src/com/dotComIt/learnWith/vo/Use
```

The only classes we need to import are the **UserModel** and the **UserVO**. We'll use the **UserModel** to create an instance that we can run tests against. We'll use the **UserVO** to control the user that we create.

Start the test suite with a **describe()** statement:

```
describe('User Model', function () {
});
```

Then create an instance of the **UserModel**:

```
let userModel: UserModel;
```

Now, create a **beforeEach()**:

```
beforeEach(() => {
    userModel = new UserModel();
    userModel.user = {userID: 24, username:
'something',
                      password: 'something',
role: userModel.TASKER_ROLE}
                     as UserVO
});
```

The code creates an instance of the **UserModel**, sets the **user** property to a new user object and that is it. All our tests will be based on this same fake user.

I'm going to use a set of embedded **describe()** statements, one for each function in the module. Since both **isUserInRole()** and **validateUser()** return Boolean values, we just need a true test and a false test for each one.

Start with **isUserInRole()**. First, create the **describe()** block:

```
describe('isUserInRole()', function () {
});
```

Create the true test:

```
it('User is in Role', () => {
    var result :boolean =
userModel.isUserInRole(userModel.TASKER_ROLE);
    expect(result).toBeTruthy();
});
```

This calls **isUserInRole()** function and checks that the result is true. It uses a new testing function, **toBeTruthy()**. This means that the test should pass if the result would return true in a Boolean condition. The false test is similar, but opposite:

```
it('User is not in Role', () => {
    var result :boolean =
userModel.isUserInRole(userModel.CREATOR_ROLE);
    expect(result).toBeFalsy();
```

```
});
```

Here we see the **toBeFalsy()** test used to compare the test. In both cases we know the user roles before-hand because we set them up in the **beforeEach()** block.

The **validateUser()** tests use a similar approach. Start with the **describe()** block:

```
describe('validateUser()', function () {
});
```

Create the test to check that the user is logged in:

```
it('User Logged In', () => {
    var result :boolean =
userModel.validateUser();
    expect(result).toBeTruthy();
});
```

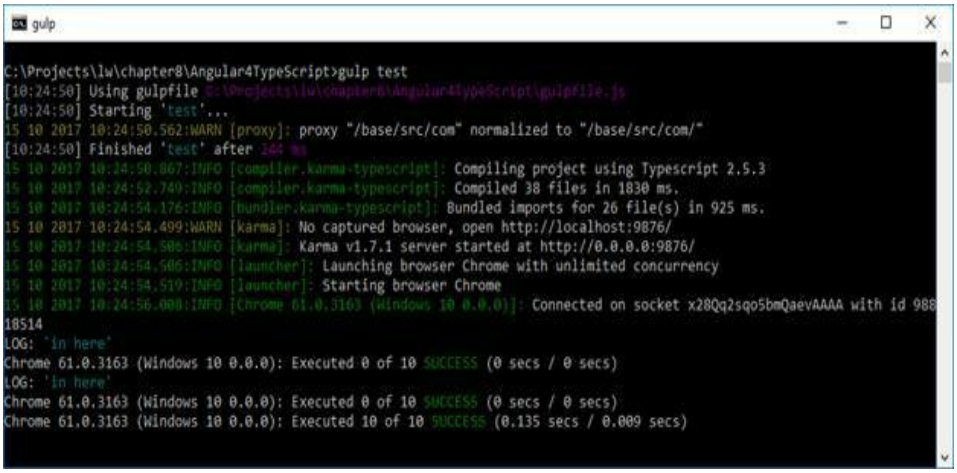
It calls the **validateUser()** function on the **userModel** instance, then checks the **result** using **toBeTruthy()**.

The “user not logged in” function has one major change:

```
it('User Not Logged In', () => {
    userModel.user = null;
    var result :boolean =
userModel.validateUser();
    expect(result).toBeFalsy();
});
```

It nulls out the **userModel**’s user object before calling the function. This is how the **validateUser()** determines if the user is logged in or not. If the user object has a value, they are logged in. Otherwise they are not logged in. It grabs the result of the function call and runs it through a **toBeFalsy()** flag.

Rerun your tests to make sure everything is working well:



The **toBeTruthy()** and **toBeFalsy()** methods should be considered helper methods. Sometimes in programming you don't have explicit values, especially when creating conditional statements. Not true may not be the same as false; but it is usually good enough for programming purposes.

Your First Angular Tests

This section will write the first Angular tests. It will talk about the **TestBed**, a testing module that will mock our app's module for testing purposes. We'll also set up a mock router.

Create a Mock Router

First, we're going to create a mock router. This will be very similar to the app's existing router with just a few changes. Let's start by reviewing the router. Open the **routing.module.ts** from the **src/com/dotComIt/learnWith/nav** directory:

```
import { NgModule }from '@angular/core';
import { RouterModule, Routes }   from
'@angular/router';
import { LoginComponent } from
"../views/login/login.component";
import { TasksComponent } from
"../views/tasks/tasks.component";

const ROUTES : Routes = [
    { path: 'login',  component: LoginComponent
},
    { path: 'tasks',  component: TasksComponent
},
    { path: '', redirectTo: 'login', pathMatch:
'full' },
    { path: '**', redirectTo: 'login' }
];
@NgModule({
    imports: [ RouterModule.forRoot(ROUTES) ],
    exports: [ RouterModule ]
})
export class AppRoutingModule {}
```

This imports the **NgModule** from the Angular Core library. It imports the **RouterModule** and **Routes** class from the Angular

Router library. It brings in our own custom views, the **LoginComponent** and **TasksComponent**. It creates a constant to contain the routing information, then defines the module with metadata. The **RouterModule** is set up using the **forRoot()** method with the **ROUTES** constant passed in. Finally, the class is exported. The application's main module loads this module and sets it up to handle routing.

Let's re-create this to use a testing **RouterModule** instead of the real thing. Create a file named **routing.module.mock.ts** in **tests/com/dotComIt/learnWith/nav**. First, do the imports:

```
import { NgModule } from '@angular/core';
import { RouterTestingModule } from
'@angular/router/testing';
import { Routes } from '@angular/router';
```

The **NgModule** and **Routes** are imported identically. But, the **RouterModule** from **@angular/router** is replaced with a **RouterTestingModule** from **@angular/router/testing**. The testing components give us the same API as the real components with modified functionality that is more apt for non-browser based testing.

Now import our view components:

```
import {LoginComponent} from

"../../../../../src/com/dotComIt/learnWith/views/
import {TasksComponent} from

"../../../../../src/com/dotComIt/learnWith/views/
```

One downside of keeping tests and application code separate is the directory traversal from one to the other. Now, create the **ROUTES** constant:

```
const ROUTES : Routes = [
    { path: 'login',  component: LoginComponent
},
    { path: 'tasks',  component: TasksComponent
```

```
},
    { path: '', redirectTo: 'login', pathMatch:
'full' },
    { path: '**', redirectTo: 'login' }
];
```

This hasn't changed a bit from the original to our mock. Create the **@NgModule**:

```
@NgModule({
    imports: [
RouterTestingModule.withRoutes(ROUTES) ],
    exports: [ RouterTestingModule ]
})
```

The exports are set up to load the **RouterTestingModule** instead of the **RouterModule**. There is an important change to the imports. Instead of using the **forRoot()** function, we use the **withRoutes()** function. Finally, export the class:

```
export class AppRoutingModule {}
```

The class is named identically in both the mock case and the application code. This is so one is easily replaceable by the other when we configure the **TestBed** module.

Configure the TestBed

The Angular testing code includes something called a **TestBed**. This is a module created for testing other modules and is considered the foundation of all Angular tests. We're going to use our **base.test.ts** file to set up the **TestBed** to parallel the main application.

First, let's review the **app.module.ts** from **src/com/dotComIt/learnWith/main**. This file contains the main application's module definition which we will mirror in the **TestBed** configuration. The module definition takes up a lot of space, so let's set it up in parts. First, we can review the imports. Start with the main libraries for Angular, ngx-datatable, and ng-bootstrap:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-
```

```
browser';
import { HashLocationStrategy, LocationStrategy }
from '@angular/common';
import { FormsModule }   from '@angular/forms';
import { NgxDatatableModule } from
'@swimlane/ngx-datatable';
import { NgbModule } from '@ng-bootstrap/ng-
bootstrap';
```

Next all the application's components are imported:

```
import { AppRoutingModule }     from
'../nav/routing.module';
import { AppComponent }  from './app.component';
import { LoginComponent} from
"../views/login/login.component";
import { TasksComponent} from
"../views/tasks/tasks.component";
import { TaskGrid } from
"../views/tasks/taskgrid.component";
import { TaskFilter } from
"../views/tasks/taskfilter.component";
import { TaskCU } from
"../views/tasks/taskcu.component";
import { TaskScheduler } from
"../views/tasks/taskscheduler.component";
```

Notice that the **AppRoutingModule** is included in this section even though it is implemented as a separate module and not a UI component. When configuring our **TestBed**, we'll ignore that and replace it with the mock **AppRoutingModule**.

Now, import the model components:

```
import { UserModel } from "../model/usermodel";
import { TaskModel } from "../model/taskmodel";
```

And finally, import the services:

```
import { AuthenticationService } from
"../services/mock/authentication.service";
import { TaskService } from
```

```
"../services/mock/task.service";
```

There are a lot of things to import here, and most will be replicated as part of the **base.test.ts** file.
Next define the **@ngModule** metadata:

```
@NgModule({
  imports:      [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    NgxDatatableModule,
    NgbModule.forRoot()
  ],
  declarations: [
      AppComponent,
      LoginComponent,
      TasksComponent,
      TaskGrid,
      TaskFilter,
      TaskCU,
      TaskScheduler
  ],
  providers : [
      {provide: LocationStrategy,
useClass:HashLocationStrategy},
      AuthenticationService,
      UserModel,
      TaskModel,
      TaskService
  ],
  bootstrap:    [ AppComponent ],
  entryComponents: [TaskCU]
})
```

The imports section loads all modules from ng-bootstrap, ngx-datatables, and Angular. It also includes our **AppRoutingModule**. Separating code into its own module is considered the best encapsulation approach, which is why all the main libraries use it. I like to do that when I'm optimizing for reuse, but do not usually do

that when building application specific components.

Finally, the **AppModule** is exported:

```
export class AppModule { }
```

If you've gone through our main book, you've seen this all before. The ColdFusion and NodeJS implementations have slightly different import statements for the services, but otherwise no changes.

Let's turn our attention back to the **base.test.ts** file in the **tests** directory root. We already put a bunch of imports in there to load libraries for CoreJS and ZoneJS libraries. Let's start by importing the **TestBed**:

```
import { TestBed } from "@angular/core/testing";
```

We're also going to load two other modules: **BrowserDyanmicTestingModule** and **platformBrowserDynamicTesting**:

```
import { BrowserDynamicTestingModule,
platformBrowserDynamicTesting }
from "@angular/platform-browser-dynamic/testing";
```

These two are used to mock how the main **AppComponent** is loaded. In our **main.ts** file from **src/com/dotComIt/learnWith/main**, we do this:

```
platformBrowserDynamic().bootstrapModule(AppModul
```

In the **base.test.ts** file, we'll do this:

```
TestBed.initTestEnvironment(BrowserDynamicTesting

platformBrowserDynamicTesting());
```

Put this line in the file right after the imports.

Now, we can add the other imports. Start with the external libraries:

```
import { HashLocationStrategy, LocationStrategy }
from '@angular/common';
```

```
import { FormsModule }   from '@angular/forms';
import { NgxDatatableModule } from
'@swimlane/ngx-datatable';
import { NgbModule} from '@ng-bootstrap/ng-
bootstrap';
```

Compared to our application code, we are skipping two imports the **BrowserModule** and **NgModule**. We don't need them for testing purposes. Now add in our main application's components:

```
import { AppRoutingModule } from

'./com/dotComIt/learnWith/nav/routing.module.mock
import { AppComponent } from

 '../src/com/dotComIt/learnWith/main/app.componen
import { LoginComponent} from

'../src/com/dotComIt/learnWith/views/login/login.
import { TasksComponent} from

'../src/com/dotComIt/learnWith/views/tasks/tasks.
import { TaskGrid} from

'../src/com/dotComIt/learnWith/views/tasks/taskgr
import { TaskFilter} from

'../src/com/dotComIt/learnWith/views/tasks/taskfi
import { TaskScheduler} from

'../src/com/dotComIt/learnWith/views/tasks/tasksc
import { TaskCU} from

'../src/com/dotComIt/learnWith/views/tasks/taskcu
```

This imports the mock **AppRoutingModule** instead of the real one, but all other imports load the same interface components.

Next load the model classes:

```
import { UserModel} from
```

```
"../src/com/dotComIt/learnWith/model/usermodel";
import { TaskModel} from
"../src/com/dotComIt/learnWith/model/taskmodel";
```

And finally, the two services:

```
import { AuthenticationService} from

"../src/com/dotComIt/learnWith/services/mock/auth
import { TaskService} from

"../src/com/dotComIt/learnWith/services/mock/task
```

If you care about testing JSONP or REST services, we'll cover that later in this chapter when we write tests against NodeJS and ColdFusion respectively.

Now, we can configure the **TestBed** by calling **configureTestingModule**:

```
beforeEach(() => {
  TestBed.configureTestingModule({
    imports : [AppRoutingModule, FormsModule,
               NgbModule.forRoot(),
NgxDatatableModule],
    declarations: [ AppComponent, LoginComponent,
TasksComponent,
                    TaskFilter, TaskGrid,
TaskScheduler, TaskCU ],
    providers : [{provide: LocationStrategy,
useClass:HashLocationStrategy},
                 AuthenticationService,
UserModel,TaskModel, TaskService
        ]
  }).overrideModule(BrowserDynamicTestingModule,
{
      set: {
          entryComponents: [ TaskCU ]
      }
  })
});
```

First thing to notice, that I put the **TestBed** configuration in a **beforeEach()** function. What is **beforeEach()**? It is a special function that is part of the **Jasmine** testing framework, similar to **describe()** and **it()**. The function allows you to run code before tests are executed. We use it to create and configure the testing module which will be used by most of our other unit tests. You can have multiple **beforeEach()** functions if needed, but here we only need one. There is also an **afterEach()** function if you want to run code after each test is executed, but I rarely use that in practice.

The **configureTestingModule()** accepts a configuration object which sets up **imports** for other modules, **declarations** for components, and **providers** for services. This is all like the **@NgModule** metadata in our main application. The **configureTestingModule()** does not support **entryComponents,** unfortunately. We used the **entryComponents** metadata to set up a component that Bootstrap could use to create the modal for editing or creating tasks. Thankfully there is a workaround. We daisy chain an **overrideModule()** method after the **configureTestingModule** is created to add the **entryComponents** metadata.

That's all we need in the **base.test.ts** file. Notice there is no formal export of a class. It isn't needed, since no other classes will use this explicitly. Based on the compilation order set in the Gulp task file list, the **base.test.ts** file will always be compiled first.

Test the AppComponent

Let's write our first Angular tests. We're going to start with the **AppComponent.ts** from **src/com/dotComIt/learnWith/main**. Let's review this file since it is simple:

```
import { Component } from '@angular/core';
@Component({
  selector: 'lw-app',
  template: `<router-outlet></router-outlet>`,
})
export class AppComponent  {  }
```

The component does nothing but load the **router-outlet**. The **router-outlet** is a special tag as part of Angular's routing module.

The routing code will look at the URL and decide which component to load its place. There isn't much to this component, so what do we want to test? I decided to test two items. The first is to validate that the component is actually loaded. The second test is to make sure that the default view is loaded based on the URL.

How do we start? Let's start with some imports. First a bunch of the Angular testing components:

```
import { async, ComponentFixture, TestBed }
from '@angular/core/testing';
```

The **async** import is used to run **beforeEach()** or **it()** blocks inside an async test zone that hides the mechanics of asynchronous processing. The **ComponentFixture** is used to create an instance of a component so you can access all properties, methods, and DOM elements of its HTML template. We already covered **TestBed**, which is a module just for testing purposes. We configured the **TestBed** in the **base.test.ts** file.

Now, let's import the **Router** and **Location** Angular libraries:

```
import {Router} from "@angular/router";
import {Location} from "@angular/common";
```

We'll use these to access the router and current, simulated, URL of the page. Finally, import the **AppComponent**:

```
import { AppComponent } from

'../../../../../src/com/dotComIt/learnWith/main/a
```

This is the one we're testing.

Now, create a **describe()** function block:

```
describe('AppComponent', function () {
});
```

All our code for testing the **AppComponent** will go inside this describe block.

Let's create some variables that we need for testing:

```
let comp: AppComponent;
let fixture: ComponentFixture<AppComponent>;
let location: Location;
let router: Router;
```

The **comp** will hold the component instance. The **fixture** is a reference to the **TestBed**'s component wrapper. The **location** and **router** will be used to access, or set, the URL.

All these variables will be given values inside a **beforeEach()** block:

```
beforeEach(async(() => {
    TestBed.compileComponents().then(() => {
        fixture =
TestBed.createComponent(AppComponent);
        comp = fixture.componentInstance;

        router = TestBed.get(Router);
        router.initialNavigation();
        location = TestBed.get(Location);
    });
}));
```

The **beforeEach()** function is defined using an **async()** call. On the **TestBed**, we tell it to compile all the components with the **compileComponents()** method. This will return a promise, and after it completes we jump into a result function. The result function gets the **fixture** by calling the **createComponent()** method on the **TestBed**, and then populates the **comp** variable with the **fixture**'s **componentInstance**.

We use the **TestBed**'s get method to retrieve the **router** and **location** instances. Also call the **initialNavigation()** on the router method to initialize it. Now let's write some tests.

The first test is to make sure that the **AppComponent** is defined:

```
it('should create component', () =>
expect(comp).toBeDefined() );
```

This uses the **expect().toBeDefined()** test. As long as the **comp** value is defined, this test will pass. I often put one of these in each

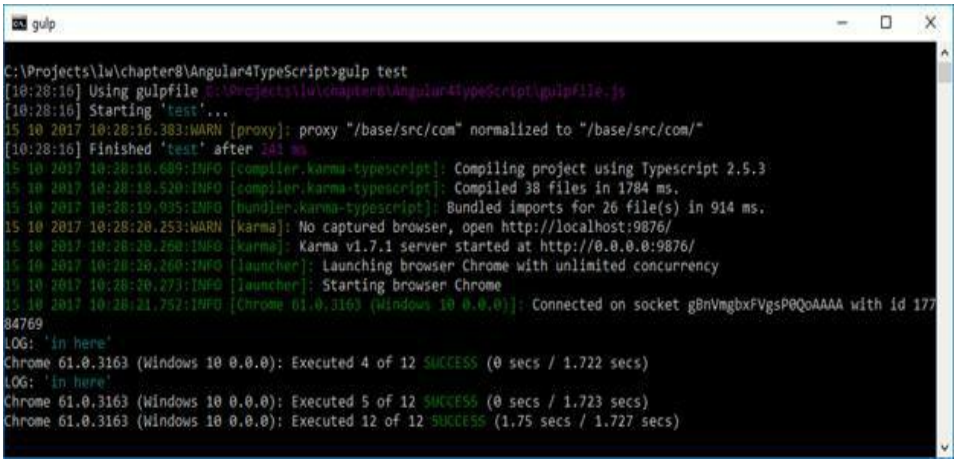
component test.

Now, let's test the default navigation:

```
it('default router navigation', () => {
    expect(location.path()).toBe('/login');
});
```

Based on the Routes we set up in our mock module, the default route should always redirect to the '/login' URL.

Rerun the tests and you should see:



The tests are successfully run.

Test the Routing Module

What do we want to test with regards to the routing module? Here is what I decided:

- The router component is created
- If the user navigates to "" then they are redirected to "/login"
- If the user navigates to "/login" then the login page loads
- If the user navigates to "/tasks" without being logged in, then they are redirected to "/login"
- If the user navigates to "/tasks" after being logged in, then the "/tasks" is loaded.

There is some replication in these tests. You can find all the code in

our source repository, but I'm going to skip the first test, since we already saw how to test that a component is created.

Let's start by looking at the imports. Start here:

```
import { async, TestBed, fakeAsync, tick } from
'@angular/core/testing';
```

This imports various testing libraries. The previous section explained the **async** and **TestBed** imports. The **fakeAsync** import will run a test inside a fake asynchronous test zone that enables linear control over asynchronous calls. It is used in conjunction with **tick** which simulates the passing of time in the fake asynchronous zone.

Now, import some of the libraries:

```
import {Router} from "@angular/router";
import {Location} from "@angular/common";
import {Observable} from "rxjs/Observable";
import 'rxjs/add/observable/of';
```

We used the **Router** and **Location** in the previous set of tests. The **Observable** and **of** imports allow us to mock some of the service calls that happen when the "/tasks" route is loaded. The app immediately starts to load the task category array and the initial load of tasks.

Our final imports focus on our own components:

```
import {AppComponent} from

"../../../../../src/com/dotComIt/learnWith/main/a
import {TaskService} from

"../../../../../src/com/dotComIt/learnWith/servic
import {UserModel} from

"../../../../../src/com/dotComIt/learnWith/model/
import {ResultObjectVO} from

"../../../../../src/com/dotComIt/learnWith/vo/Res
```

We bring up the **AppComponent** which is needed for the **router-outlet**. The **TaskService** will be used to intercept service calls. **UserModel** will be used to intercept the call to check if a user is logged in. The **ResultObjectVO** will be used as the return value from our intercepted service calls. We haven't learned how to intercept service calls yet, but I'll explain it when it comes up.

Let's move onto the Unit Tests. Start with the describe block:

```
describe('Routing Module', function () {
});
```

We've seen this before. Then define the variables we need:

```
let location: Location;
let router: Router;
let taskService : TaskService;
let userModel : UserModel;
```

After the initial creation of the **AppModule** we don't need to reference it, so we don't have a global variable for the **ComponentFixture** or **AppModule** itself.

Now, add a **beforeEach()**:

```
beforeEach(async(() => {
    TestBed.compileComponents().then(() => {
        TestBed.createComponent(AppComponent);
        router = TestBed.get(Router);
        router.initialNavigation();
        taskService = TestBed.get(TaskService);
        userModel = TestBed.get(UserModel);
        location = TestBed.get(Location);
    });
}));
```

First the **compileComponents()** method is called on the **TestBed**. This returns a promise, which is then resolved, and a success function is run. The success function creates the **AppComponent**, saves the **router, taskService**, **userModel**, **location**, and then initializes the **router**. We're just saving a lot of values for the next step.

Let's start with a simple test. If the user navigates to "", or nothing, then we redirect them to "/login":

```
it('navigate to "" redirects you to /login',
fakeAsync(() => {
    router.navigate(['']);
    tick();
    expect(location.path()).toBe('/login');
}));
```

This is where the **fakeAsync()** and **tick()** code comes in. Under the hood the router uses asynchronous code, so our test must be run in the fakeAsync test zone. The **tick()** method tells us that our **fakeAsync** call has completed. So, first we call **router.navigate()** to direct the user to the empty URL. This will most likely be the state of the app when it first loads. Then the **tick()** is called which should force the router to redirect to '.login'. Then we use the **toBe** test. I love that the unit testing commands are written to be self-explanatory. Here is it in isolation:

```
expect(location.path()).toBe('/login');
```

I expect **location.path()** to be '/login'. Feel free to rerun your test and it should work.

Let's create our first embedded **describe()**:

```
describe('navigate to "tasks"', function () {
});
```

We have two different conditions for navigating to the "/tasks" route. If the user is logged in, then the path should stay at "/tasks". If the users is not logged in, the user should be redirected to "/login". This logic is defined inside the **tasks.component.ts** file at **src/com/dotComIt/learnWith/views/tasks**:

```
if ( !this.userModel.validateUser()) {
    this.router.navigate(['/login']);
}
```

The code exists inside the **ngOnInit()** method, which is executed as part of Angular's life cycle. So we don't have to deal with actually

logging a user in, we'll want to intercept the **validateUser()** method to return the proper value.

Let's define a **beforeEach()**:

```
beforeEach(() => {
  spyOn(taskService, 'loadTaskCategories').and

.returnValue(Observable.of(ResultObjectVO));
  spyOn(taskService, 'loadTasks').and

.returnValue(Observable.of(ResultObjectVO));
});
```

This **beforeEach()** uses a new command, **spyOn().** he **spyOn()** method is used to watch an object for a method call. You can use this in tests to intercept the method and return specific values. Alternatively, you could use it to verify that a certain method was called. The first line the **spyOn()** is looking at the **taskSevice** as the first argument, and the **loadTaskCategories()** method as the second argument. New function calls are daisy-chained after it, which make things read as simple English. Spy on the **taskService** and look for calls to the **loadTaskCategories()** method. When that happens, return an **Observable** of **ResultObjectVO**. For these tests we don't care about the returned value, we just want to prevent the **loadTaskCategories** method from being called. We do the same thing for the **loadTasks()** method.

Now let's see the first test:

```
it(' works because user is logged in',
fakeAsync(() => {
    spyOn(userModel,
'validateUser').and.returnValue(true);
    router.navigate(['tasks']);
    tick();
    expect(location.path()).toBe('/tasks');
}));
```

We're using a different daisy chained methods with the **spyOn()** method here. That is because we care about the returned value.

We are spying on the **userModel**. When the **validateUser()** method is called, the value true should be returned. This signifies that the user is logged in.

Then we route the user to the tasks route, call the **tick()** function to execute our async calls in the fakeAsync test zone, and then we check the results. The **location.path()** should be '/tasks'. Easy, right?

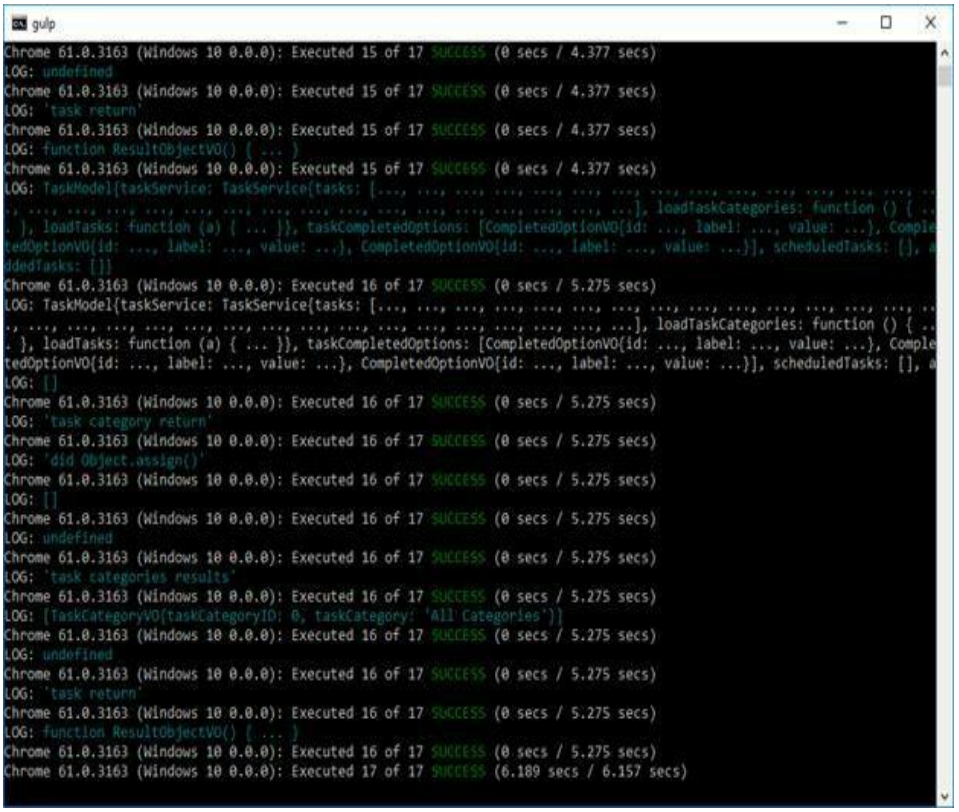
I had a bit more trouble with the situation where the user is not logged in and had to tackle it in a different way:

```
it(' does not work because user is not logged
in', fakeAsync(() => {
    spyOn(userModel,
'validateUser').and.returnValue(false);
    router.navigate(['tasks']);
    spyOn(router,'navigate').and.returnValue({});
    tick();

expect(router.navigate).toHaveBeenCalledWith(['/l
}));
```

First, the test uses **spyOn()** to make sure that the **validateUser()** function returns false. This signifies that the user is not logged in, so when the **NgOnInit()** method executes in the tasks component, the user will be redirected to the '/login' route.

The second line of test calls the **router.navigate()** to direct the user to the 'tasks' route. If you call the **tick()** method immediately, you'll get errors related to timers in the queue. This is because under the hood a **setTimeout()** is called, and **setTimeout()** does not work in the fakeAsync test zone. I got around this by immediately using **spyOn()** to replace the **router**'s **navigate()** function. That intercepts the actual redirect and avoids the **setTimeout()** calls. Then we call **tick()** to execute the async calls, which should redirect the user. Since we intercepted the call, we cannot check the **location.path()** like we did for the previous test. So, we instead look at the **spyOn()** results to make sure that **router.navigate()** was called with the argument "/login".

Rerun your tests and you should see something like this:



If you're following this chapter closely, I probably have more tests than you have. That is because I'm focusing this chapter on new things, not showing tests that just mimic what you've already learned. Unfortunately, all the **console.log()** statements in the code will output it's details to the console. Aside from those two issues, we have some unit testing scripts to successfully test the routing module.

Test Events

An important part of Angular development is making use of events. This is how one component tells its parent that something happened. A common event example is a button click, and then the button will execute some code inside the component that contains the button. When you start to build components with encapsulation in mind, then you realize that having all code contained in a single component is not always best. The component that triggers the event is not always the best component to handle the event. Event emitters help a child component communicate with its parent.

Review the TaskFilter

Our **TaskFilter** component makes use of events. Open up the **taskfilter.component.ts** file from the **src/com/dotComIt/learnWith/views/tasks** directory. This component contains various options for filtering the main task grid, but has no access to the **TaskGrid** component. This component also contains a "New Task" button, but does not contain code to create the new task. To filter the grid, or create a new task, the **TaskFilter** will emit events for the **Tasks** component to process.

Let's look at the code that makes this happen. I'm not going to share the full **TaskFilter** component, just the relevant parts to events. First, the imports:

```
import { EventEmitter, Output} from
"@angular/core";
```

The **EventEmitter** is the class used to create the events. The **Output** is metadata used to define events that this component dispatches. Inside the component we create our two outputs:

```
@Output() filterRequest = new
EventEmitter<TaskFilterVO>();

@Output() newTaskRequest = new EventEmitter();
```

Now the parent component can listen for these events. Open the **tasks.component.html** in the **src/com/dotComIt/learnWith/views/tasks** directory to see this in action:

```
<taskfilter class="taskFilter"

(filterRequest)="filterRequest($event)"
            (newTaskRequest)="newTask()">
</taskfilter>
```

The **Tasks** component listens for the events, and runs its own functions when the events are emitted.

Back to the **TaskFilter**, this is the code that emits the **newTaskRequest**:

```
newTask() : void {
    this.newTaskRequest.emit();
}
```

That one is easier since it does not have an extra object attached to it. Look at the **filter()** method for the **filterRequest()** event:

```
filter():void {
    let taskFilter : TaskFilterVO = new
TaskFilterVO();
    // Stuff to populate taskFilter instance
    this.filterRequest.emit(taskFilter);
}
```

It uses the same **emit()** function on the **EventEmitter** instance, but passes an object to it. All this works together so that our **TaskFilter** component can communicate with the **Tasks** component. But, how do we test it?

Write the Tests

The first thing we need for our tests is to import them:

```
import {async, TestBed,ComponentFixture} from
'@angular/core/testing';
import {TaskFilter} from
```

```
"../../../../../../src/com/dotComIt/learnWith/views/tas
```

This imports the Angular testing components, which we've seen before, and the **TaskFilter** component that needs to be tested.

```
describe('TaskFilter Component', function () {
});
```

Create the variables that allow us to get a hook into the component:

```
let fixture: ComponentFixture<TaskFilter>;
let comp: TaskFilter;
```

The compile the components on the **TestBed** in a **beforeEach()** block:

```
beforeEach(async(() => {
    TestBed.compileComponents().then(() => {
        fixture =
TestBed.createComponent(TaskFilter);
        comp = fixture.componentInstance;
    });
}));
```

The tests themselves are simple:

```
it('Emit FilterRequest', function () {
    spyOn(comp.filterRequest,"emit")
    comp.filter();

expect(comp.filterRequest.emit).toHaveBeenCalled(
});
```

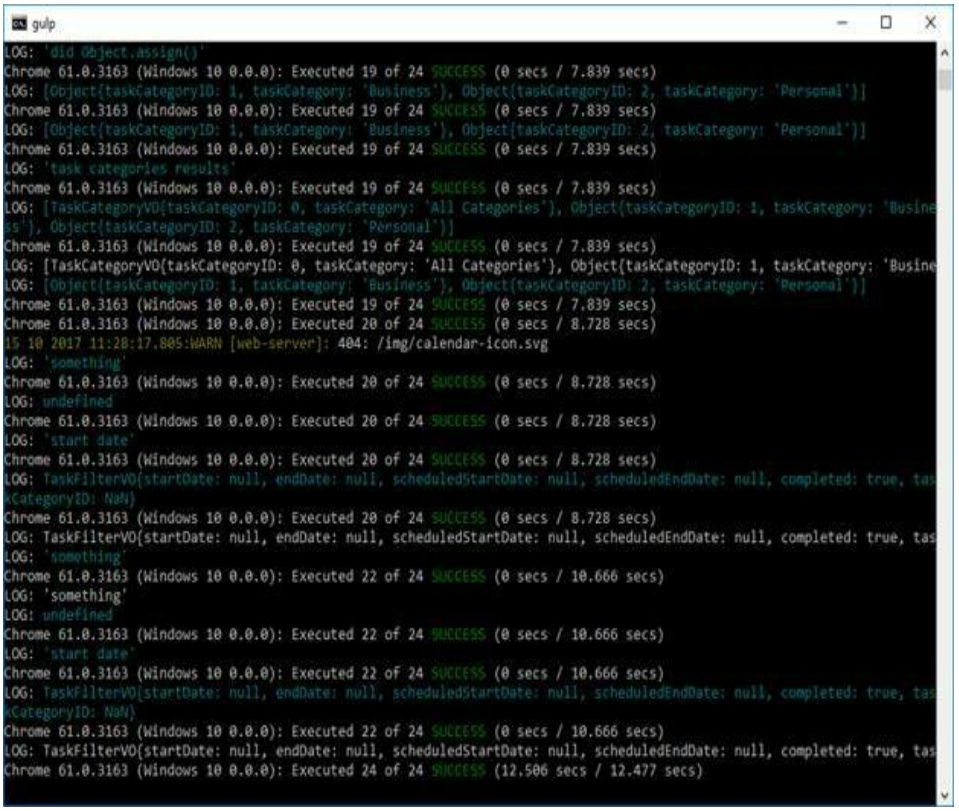
I use **spyOn()** to intercept the **emit()** function on the component's **filterRequest** instance. Then I call the **filter()** method on the **comp**. Then the test expects that **comp.filterRequest.emit()** should have been called. In this case, I'd prefer to use a **toHaveBeenCalledWith()**, however since the **TaskFilterVO** instance is private to the method, we have no way to access it to validate the object's values.

The code for checking the **newTask()** function is almost identical:

```
it('Emit NewTask', function () {
     spyOn(comp.newTaskRequest,"emit")
     comp.newTask();

expect(comp.newTaskRequest.emit).toHaveBeenCalled
});
```

It uses **spyOn()** with the **newTaskRequest** instead of **filterRequest**, and calls the **newTask()** method on the **comp** instead of **filter()**. But, the concept behind the two tests is the same.

Rerun your tests:



No problems, we are good to move on.

Test a Bootstrap Popup

To create a new task, or edit an existing task, the application creates a modal from the ng-bootstrap library. How do we test that? This section will show you how.

Review the Code to Create the Popup

First, take a look at the **tasks.component.ts** file in **src/com/dotComIt/learnWith/views/tasks**. Find the **openTaskWindow()** method:

```
private openTaskWindow(title:string, task:TaskVO
= null) {
  const modalRef = this.modalService.open(TaskCU
);
  modalRef.componentInstance.title = title;
  modalRef.componentInstance.task = task;
  modalRef.result.then((result) => {
   if (!task) {
     this.taskgrid.tasks.push(result[0]);
   } else {
     for (let index = 0; index <
this.taskgrid.tasks.length; ++index) {
      if (this.taskgrid.tasks[index].taskID ===
result[0].taskID) {
        this.taskgrid.tasks[index].description =
result[0].description;
        this.taskgrid.tasks[index].taskCategory =
result[0].taskCategory;
        this.taskgrid.tasks[index].taskCategoryID
= result[0].taskCategoryID;
        break;
      }
    }
  }
  }).catch( (result) => {
      console.log('cancelling changes');
  });
```

```
};
```

A **modalRef** variable is created. This is a reference to the modal. It is created by calling the **open()** method on the **modalService**. The **modalService** comes from the ng-bootstrap library and is injected into the **TasksComponent** as part of the constructor using Angular's dependency injection syntax, like this:

```
constructor(private modalService: NgbModal
            // Other Injections
){}
```

The **modalRef** as a promise that will execute when the modal is closed. This code does nothing when the modal is dismissed, but if it is closed, then the new task will be added to the **taskgrid.tasks** array. If it is updated, then task data will be updated in the **taskgrid.tasks** array.

Notice that the method is private. In a strongly typed language this would make it hard to test, since TypeScript transpiles to JavaScript, where no form of private exists, we are able to test this without changing the code. Your IDE may complain about it, though.

Test the Code to Create the Popup

Create a file named **tasks.component.test.ts** in the **tests/com/dotComIt/learnWith/views/tasks** directory. Start with the imports:

```
import {async, TestBed, ComponentFixture} from
'@angular/core/testing';
import {NgbModal, NgbModalRef} from "@ng-
bootstrap/ng-bootstrap";
```

First, we imported the required Angular testing components. We also brought in a few components from the ng-bootstrap library required to make the modal.

Next, import the components we'll need:

```
import {AppComponent} from

"../../../../../../src/com/dotComIt/learnWith/mai
```

```
import {TasksComponent} from

"../../../../../../src/com/dotComIt/learnWith/vie
import {TaskCU} from
 "../../../../../../src/com/dotComIt/learnWith/vi
```

The **AppComponent** will be used to initialize the router-outlet, but we won't reference it inside the test. The **TasksComponent** is the one with the code to open the modal. Finally, the **TaskCU** component is the modal.

We'll need a few value objects too:

```
import {TaskVO} from

"../../../../../../src/com/dotComIt/learnWith/vo/
```

So, we are good to go. Now, create the **describe()** block:

```
describe('Tasks Component', function () {
});
```

In the full source code for the book I do a bunch of tests against the Tasks Component, but here we will focus exclusively on the modal creation function. Create some variables:

```
let fixture: ComponentFixture<TasksComponent>;
let comp: TasksComponent;
let modalService: NgbModal;
```

There is a value for the service to create the modal, one for the **TasksComponent**, and one for the **ComponentFixture**. **ComponentFixture** is used to access the component instance and related elements, such as the component's DOM in tests.

Create a **beforeEach()** to initialize the **TestBed** and populate our variables:

```
beforeEach(async(() => {
    TestBed.compileComponents().then(() => {
        fixture =
TestBed.createComponent(TasksComponent);
        modalService = TestBed.get(NgbModal);
```

```
TestBed.createComponent(AppComponent).componentIn
        comp = fixture.componentInstance;
    });
}));
```

The only reason to initialize the **AppComponent** is so that the **router-outlet** exists, which we'll need activated later when we initialize our components. An **AppComponent** instance is not stored for later access. The other three variables are.

Now, create an embedded **describe()** for the **openTaskWindow()** function:

```
describe('openTaskWindow()', function() {
});
```

We'll create a bunch of embedded describes for this set of tests. But, first here is the first test:

```
it('Modal Opened', function () {
    spyOn(modalService,"open").and.callThrough();
    comp.openTaskWindow("Test Title");
    expect(modalService.open).toHaveBeenCalled();
});
```

We use **spyOn()** to check the **open()** method on the **modalService**. This verifies that the **open()** method is called as a result of the **openTaskWindow()** call.

Add one last embedded **describe()** function:

```
describe('Modal Ref Processing', function() {
});
```

I'll create two values unique to this **describe()** block:

```
let modalRef : NgbModalRef;
let task : TaskVO;
```

The **modalRef** will be a reference to the open modal window. The task will be a **TaskVO** instance passed into the **openTaskWindow()** function. Add a **beforeEach()**:

```
beforeEach(() => {
    modalRef = modalService.open(TaskCU );

spyOn(modalService,"open").and.returnValue(modalR
});
```

I use the **beforeEach()** to open the window, and also to spy on the open method.

Next, I created a few tests to check the **title** and **task** values on the modal popup. First, check that the empty task is null, and that the title is properly set:

```
it('Modal Opened, Title Set, no Task', function
() {
    comp.openTaskWindow("Test Title", task);
    expect(modalService.open).toHaveBeenCalled();

expect(modalRef.componentInstance.title).toBe("Te
Title");

expect(modalRef.componentInstance.task).toBeNull(
});
```

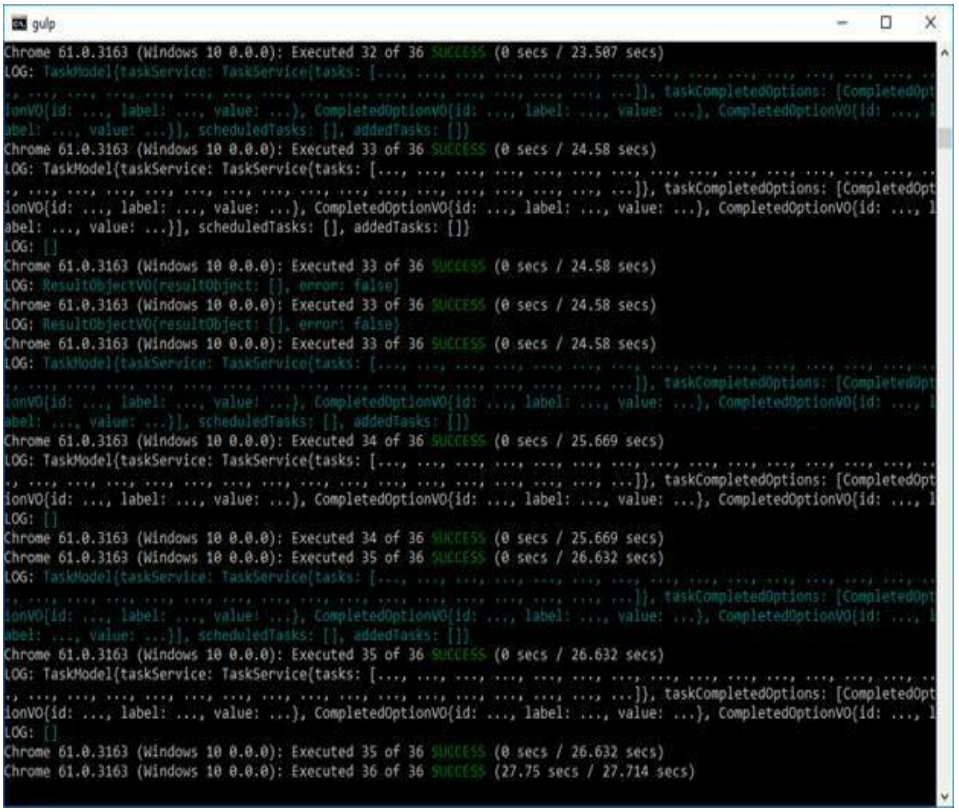
You'll notice a new test we hadn't seen yet: **toBeNull()**. This just compares the value to a null. If it is, then it returns false. Otherwise it returns true.

The test with a task is similar, just that a **TaskVO** instance is created and sent into the **openTaskWindow()** function:

```
it('Modal Opened, Task Set', function () {
    task = new TaskVO();
    comp.openTaskWindow("Test Title", task);
    expect(modalService.open).toHaveBeenCalled();

expect(modalRef.componentInstance.title).toBe("Te
Title");

expect(modalRef.componentInstance.task).toBe(task
});
```

Instead of checking for a null value on the **componentInstance.task**, it compares the set value with the value we passed in using **toBe()**.

This would be a good time to rerun your tests just to make sure everything is going well:



Things are about to get complicated because we have to deal with promise fulfillment inside of the **openTaskWindow()** function.

First, create the test:

```
it('Modal Opened, Closed with New Task', (done :
DoneFn) => {
});
```

This test will focus on closing the modal after entering data for a new task. Notice I'm passing a parameter, **done**, into the test. This is a function that tells Jasmine to run the code asynchronously. This is a special Jasmine construct, and is different than the **async** or

fakeAsync test beds. The Angular constructs have issues with any sort of timer under the hood, while the Jasmine approach does not.

First, open the modal:

```
comp.openTaskWindow("Test Title");
```

Create a **TaskVO**:

```
let task : TaskVO[] = [{ taskID : 700,description
: "something"} as TaskVO];
```

The **task** object is passed into the **close()** method of the modal reference.

Now, run this:

```
fixture.detectChanges();
```

The **detectChanges()** method forces Angular to redraw the screen. We're running this now to make sure that the default **tasks** array is loaded inside the tasks grid. The **openTasksWindow()** method will throw errors in the in line result handler if this value is not initialized.

After that completes, we need to run code:

```
fixture.whenStable().then(() => {
});
```

The **fixture.whenStable()** returns a promise and we want to make sure that the **detectChanges()** is complete before continuing our code execution. Inside the result function we can close the **modalRef**:

```
modalRef.close(task);
```

The argument into the **modalRef()** is the task we created earlier. Now, we need another check and promise:

```
fixture.whenStable().then(() => {
});
```

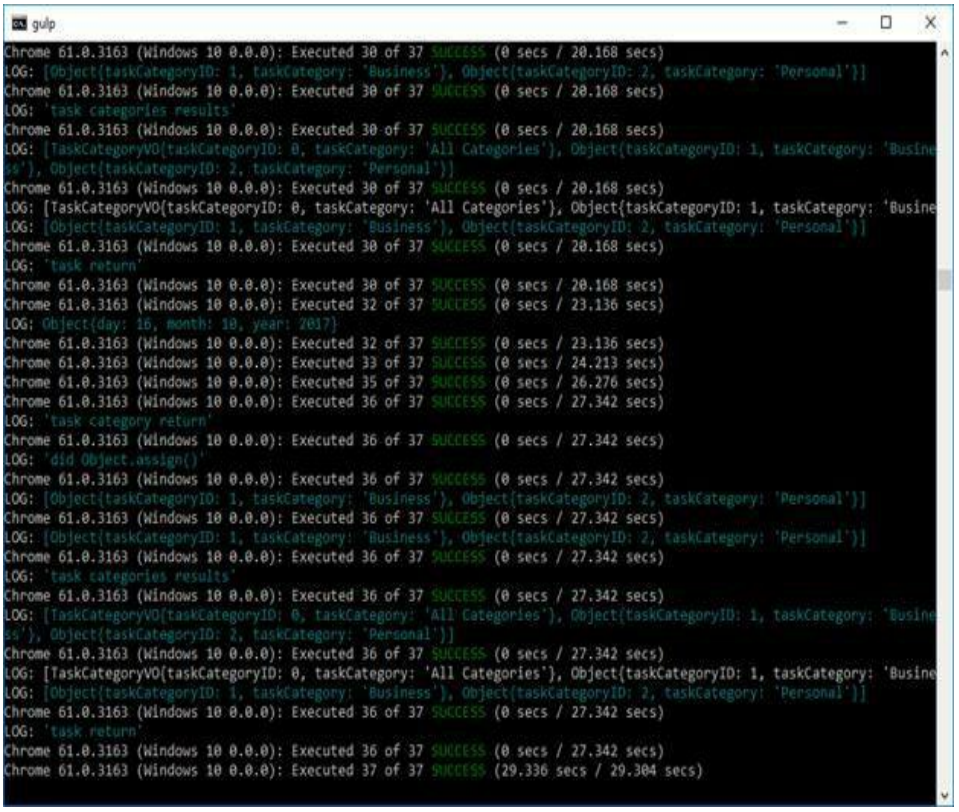
This is telling the testing code to wait until the **close()** promise

completes. It is an embedded **whenStable()** check. Inside the result handler we can run our tests:

```
expect(fixture.componentInstance.taskgrid.tasks.l
expect(fixture.componentInstance.taskgrid.tasks[

fixture.componentInstance.taskgrid.tasks.length-1
       ]).toBe(task[0]);
done();
```

The first test checks for the length of the **tasks** array inside the **taskgrid**. The default load is 24, so after adding a new item the total length should be 25. It also verifies that the last item on the task list is the task object we passed into the **close()** method. The final element after the tests is to call the **done()** method. This is how Jasmine determines that the test is complete and it is not waiting for more callbacks.

Rerun the tests:



Let's create another test for when a task is updated. We're going to use similar concepts to the new task popup, but a slightly different implementation.

First, create the test stub:

```
it('Modal Opened, Edit Existing Task', (done :
DoneFn) => {
});
```

The first thing we want to do in the test is to call **detectChanges()**:

```
fixture.detectChanges();
```

We are loading the initial tasks array so we can edit an existing task. Add the **whenStable()** code next:

```
fixture.whenStable().then(() => {
});
```

Inside this function, we'll grab a task object:

```
let task : TaskVO =
fixture.componentInstance.taskgrid.tasks[

fixture.componentInstance.taskgrid.tasks.length-
1]
                    as TaskVO;
```

Change its description:

```
task.description = "Something Else";
```

And then call the **openTaskWindow()** method:

```
comp.openTaskWindow("Test Title", task);
```

Close the window immediately:

```
modalRef.close(task);
```

Now, we come to our second **whenStable()** block:

```
fixture.whenStable().then(() => {
});
```

Inside that result function we'll execute our tests:

```
expect(fixture.componentInstance.taskgrid.tasks.l
expect(fixture.componentInstance.taskgrid.tasks

[fixture.componentInstance.taskgrid.tasks.length-
1])
        .toBe(task);
expect(fixture.componentInstance.taskgrid.tasks

[fixture.componentInstance.taskgrid.tasks.length-
1].description)
        .toBe("Something Else");
```

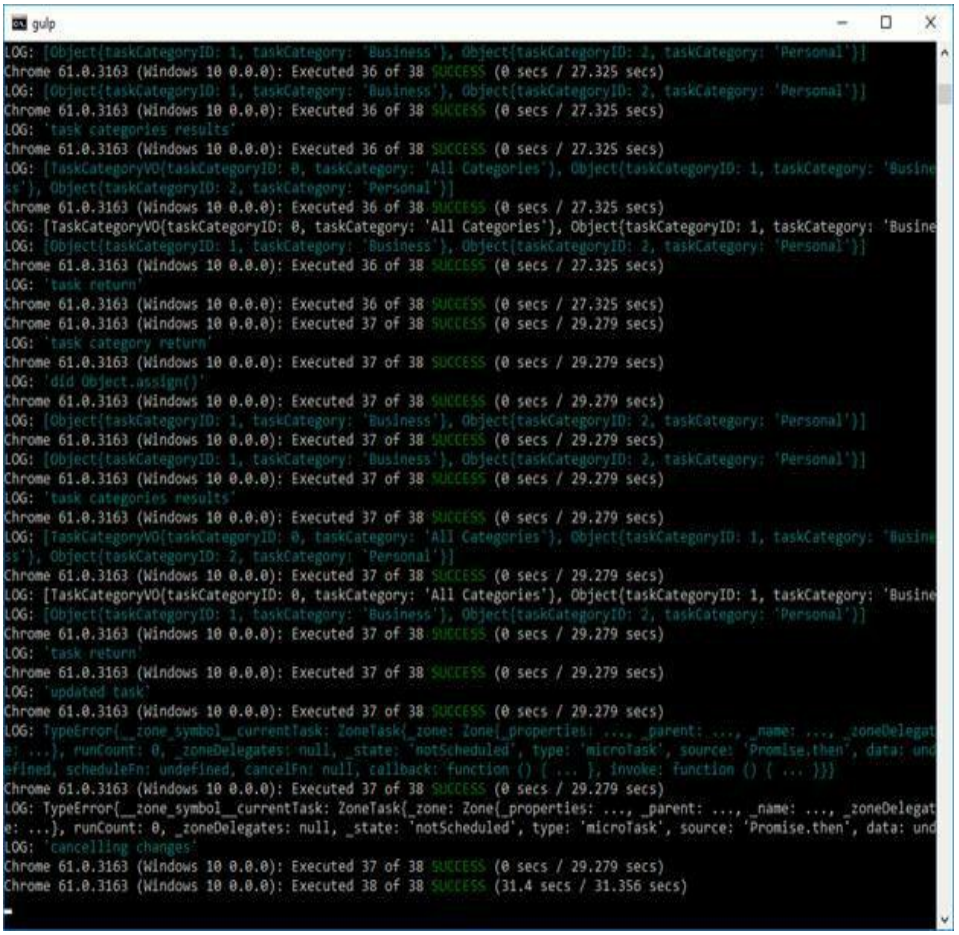
The first test verifies that the length of the tasks array has not changed. The second one makes sure that the last item on the tasks array is the same task we edited. The third task checks the value of

the description field on the task to make sure it retained our modified value.

Finally call the done function:

```
done();
```

Rerun the tests one last time:



With tests for the creation of the popup behind us, let's write some tests against the actual popup code.

Review the Popup's Code

Let's take a look at the modal code we are going to test against. Open the **taskcu.component.ts** file from the **src/com/dotComIt/learnWith/views/tasks** directory. The

important method is the **onSave()** method:

```
onSave():void {
  this.taskUpdateError = '';
  this.taskService.updateTask(this.task,
this.userModel.user).subscribe(
      (result) => {
          if ( result.error ) {
              this.taskUpdateError = 'There was a
problem saving the task.';
              return;
          }

this.activeModal.close(result.resultObject);
      },
      (error) => {
          this.taskUpdateError = 'There was a
problem saving the task.';
      }
  );
}
```

The code calls the **taskService.updateTask()** method. For a successful result, we close the modal and pass the modified task object as a parameter. For a failed call, we set the **taskUpdateError** value and stop processing the method. Even though the **onSave()** method can be used for both updating a task or creating a new task, that functionality is encapsulated away from this method so we don't have to worry about it.

Test the Popup's Code

Create a new test file named **taskcu.component.test.ts** in the **tests/com/dotComIt/learnWith/views/tasks** directory. Let's start with the imports:

```
import {async, TestBed} from
"@angular/core/testing";
import {NgbModal, NgbModalRef} from "@ng-
bootstrap/ng-bootstrap";
import {Observable} from "rxjs/Observable";
```

```
import {Observer} from "rxjs/Observer";
```

The first set of imports includes the Angular testing components. It includes the ng-bootstrap libraries to deal with the module. It also includes a few rxjs classes so we can use Observables.

Now import our own components:

```
import {TaskCU} from

"../../../../../../src/com/dotComIt/learnWith/views/tas
import {TaskService} from

"../../../../../../src/com/dotComIt/learnWith/services/
import {ResultObjectVO} from

"../../../../../../src/com/dotComIt/learnWith/vo/
import {TaskVO} from

"../../../../../../src/com/dotComIt/learnWith/vo/
```

We have the **TaskCU** component which is the component we're testing. The **TaskService** will be used with **spyOn()** to intercept remote server calls. The **ResultObjectVO** is used because that is the type of **Observable** returned from the service call. because we'll need it to compare the tasks array after updating a task. The **TaskVO** is brought in because it represents the data we'll be updating.

As always start with the **describe()** function:

```
describe('TaskCUComponent', function () {
});
```

Then define your variables:

```
let comp: TaskCU;
let taskService : TaskService;
let modalService: NgbModal;
```

It is an instance of the **TaskCU** component, the **TaskService**, and the **NgbModal**. Set values to all these in a **beforeEach()**:

```
beforeEach(async(() => {
    TestBed.compileComponents().then(() => {
        taskService = TestBed.get(TaskService);
        modalService = TestBed.get(NgbModal);
        const modalRef : NgbModalRef =
modalService.open(TaskCU);
        comp = modalRef.componentInstance;
    });
}));
```

A key thing to notice here is that instead of using **TestBed.get()** to create the **TaskCU** instance, we use the **modalService.** We are creating the popup modal in the test just as we would in the code. No TaskCU instance is created as part of the main module; the **modalService** creates it on demand. This would throw errors if we did not define the **TaskCU** as part of the **entryComponents** metadata on the **NgModule** in our **base.test.ts** file.

Let's get down to writing tests. First, write the **it()** function:

```
it('Failure', function () {
});
```

This first test will look at the failure condition, but it's blank now. Use **spyOn()** to intercept the **TaskService updateTask()** method:

```
spyOn(taskService, 'updateTask').and.returnValue(
});
```

The **updateTask()** method will return an explicit value, but we haven't defined it yet. We want to return an **Observable** instance, just like the it was a real asynchronous service call:

```
Observable.create((observer :
Observer<ResultObjectVO>) => {
    let result : ResultObjectVO = new
ResultObjectVO();
    result.error = true;
    observer.next(result);
    observer.complete();
})
```

There is nothing asynchronous in this code. It just creates a **ResultObjectVO** instance, sets the **error** property to true, then completes the **observer** with the **next()** and **complete()** functions. When we created mock services for the application, we used a similar approach for the services, except we added a timer to the mix to artificially delay the response. Here we just immediately resolve the **Observable**.

Now call the **onSave()** method:

```
comp.onSave();
```

This will trigger the service call, which is being intercepted by the **spyOn()**, which will return the **Observable** we created above, which will immediately resolve. As such we can run our assertion test immediately:

```
expect(comp.taskUpdateError).toBe('There was a
problem saving the task.');
```

This should resolve to true.

Our method to test the success method is not much different:

```
it('Success', function () {
    spyOn(taskService,
'updateTask').and.returnValue(
        Observable.create((observer :
Observer<ResultObjectVO>) => {
            let result : ResultObjectVO = new
ResultObjectVO();
            result.error = false;
            result.resultObject = new TaskVO();
            observer.next(result);
            observer.complete();
        })
    );
    comp.onSave();
    expect(comp.taskUpdateError).toBe('');
});
```

The **result.error** property Is set to false instead of true. And the

resultObject includes a new **TaskVO** instance. The final assertion is expecting the **taskUpdateError** value to be an empty string, meaning there was no error.

I haven't written about this yet, but we can also test the state when an **Obervable** fails, so let's do it. We have been resolving the **Observable** using the **next()** function. To force the **Observable** to fail, we use the **error()** function. Create a rejection tests:

```
it('Rejection', function () {
});
```

Use the **spyOn()** to intercept the **updateTask()** call:

```
spyOn(taskService, 'updateTask').and.returnValue(
    Observable.create((observer :
Observer<ResultObjectVO>) => {
        observer.error({});
        observer.complete();
    })
);
```

For this sample I didn't even bother to create a **ResultObjectVO** instance since I know it isn't needed. I passed an empty object as the argument to the **error()** function.

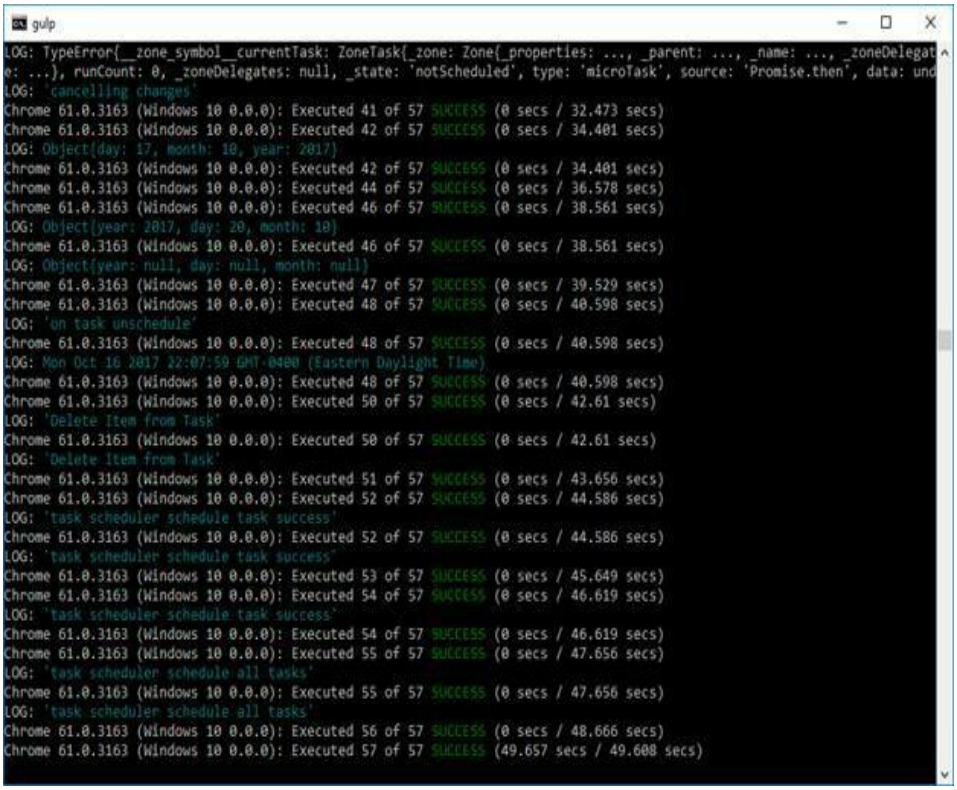
Then trigger the save method after the **Observable** code:

```
comp.onSave();
```

And finally test the assertion:

```
expect(comp.taskUpdateError).toBe('There was a
problem saving the task.');
```

Now is a good time to try running your code again:


gulp
LOG: TypeError{__zone_symbol__currentTask: ZoneTask{_zone: Zone{_properties: ..., _parent: ..., _name: ..., _zoneDelegat
e: ...}, runCount: 0, _zoneDelegates: null, _state: 'notScheduled', type: 'microTask', source: 'Promise.then', data: und
LOG: 'cancelling changes'
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 41 of 57 SUCCESS (0 secs / 32.473 secs)
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 42 of 57 SUCCESS (0 secs / 34.401 secs)
LOG: Object{day: 17, month: 10, year: 2017}
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 42 of 57 SUCCESS (0 secs / 34.401 secs)
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 44 of 57 SUCCESS (0 secs / 36.578 secs)
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 46 of 57 SUCCESS (0 secs / 38.561 secs)
LOG: Object{year: 2017, day: 20, month: 10}
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 46 of 57 SUCCESS (0 secs / 38.561 secs)
LOG: Object{year: null, day: null, month: null}
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 47 of 57 SUCCESS (0 secs / 39.529 secs)
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 48 of 57 SUCCESS (0 secs / 40.598 secs)
LOG: 'on task unschedule'
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 48 of 57 SUCCESS (0 secs / 40.598 secs)
LOG: Mon Oct 16 2017 22:07:59 GMT-0400 (Eastern Daylight Time)
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 48 of 57 SUCCESS (0 secs / 40.598 secs)
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 50 of 57 SUCCESS (0 secs / 42.61 secs)
LOG: 'Delete Item from Task'
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 50 of 57 SUCCESS (0 secs / 42.61 secs)
LOG: 'Delete Item from Task'
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 51 of 57 SUCCESS (0 secs / 43.656 secs)
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 52 of 57 SUCCESS (0 secs / 44.586 secs)
LOG: 'task scheduler schedule task success'
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 52 of 57 SUCCESS (0 secs / 44.586 secs)
LOG: 'task scheduler schedule task success'
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 53 of 57 SUCCESS (0 secs / 45.649 secs)
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 54 of 57 SUCCESS (0 secs / 46.619 secs)
LOG: 'task scheduler schedule task success'
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 54 of 57 SUCCESS (0 secs / 46.619 secs)
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 55 of 57 SUCCESS (0 secs / 47.656 secs)
LOG: 'task scheduler schedule all tasks'
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 55 of 57 SUCCESS (0 secs / 47.656 secs)
LOG: 'task scheduler schedule all tasks'
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 56 of 57 SUCCESS (0 secs / 48.666 secs)
Chrome 61.0.3163 (Windows 10 0.0.0): Executed 57 of 57 SUCCESS (49.657 secs / 49.608 secs)

Testing Services

I'm going to write tests for three types of services. First, I'll focus on the mock services which do not make real remote calls, but use a timer under the hood. Then, I'll test the ColdFusion services which use http services. Finally, I'll show the NodeJS services which use JSONP to communicate between the app and service layer. For each set of services, I'm going to focus on the **AuthenticationService**, because it is simpler than the **TaskService**, but both use similar testing concepts.

Review the Mock AuthenticationService

Find the mock authentication service in the **src/com/dotComIt/learnWith/services/mock** directory. Open up **authentication.service.ts**. We only care about the authenticate method. It uses hard coded data and a **setTimeout()** to mimic a real service call:

```
authenticate(username : string, password :
string) : Observable<ResultObjectVO>  {
  let o : Observable<ResultObjectVO> =
Observable.create(
     (observer : Observer<ResultObjectVO>) => {
         setTimeout(() => {
             let result : ResultObjectVO = new
ResultObjectVO();
             if (( username === 'me' ) && (
password === 'me' )) {
                 result.error = false;
                 result.resultObject = new
UserVO();
                 result.resultObject.userID = 1;
                 result.resultObject.username =
"me";
                 result.resultObject.role = 1;
             } else if (( username === 'wife' )
&& ( password === 'wife' )) {
```

```
                    result.error = false;
                    result.resultObject = new
UserVO();
                    result.resultObject.userID = 2;
                    result.resultObject.role = 2;
                    result.resultObject.username =
"wife";
                } else {
                    result.error = true;
                }
                observer.next(result);
                observer.complete();
            }, 1000);
        });
  return o;
};
```

It creates and returns an **Obervable** instance that returns a **ResultObjectVO** instance. Inside the **Observable**'s **create()** function, **setTimeout()** is used. The **username** and **password** are manually checked against the sample logins. If they match, a hard-coded **UserVO** object is returned as part of the **result**. The **observer.next()** and **observer.complete()** values are used to trigger the **subscribe()** method in the invoking code. It's pretty simple code, and explained in more detail in the main book.

Test the Mock Service

We can test the mock service class as if it had no dependencies on the Angular framework. First, let's start with the imports. First, import the **Observable** class:

```
import {Observable} from "rxjs/Observable";
```

That will be used for handling the service call response. Then import the **AuthenticationService** instance and the **ResultObjectVO**:

```
import {AuthenticationService} from

"../../../../../../src/com/dotComIt/learnWith/services/r
```

```
import {ResultObjectVO} from

"../../../../../../src/com/dotComIt/learnWith/vo/
```

These classes will be used to call the service and handle the results.

Create the **describe()** function:

```
describe('Mock Authentication Service', function
() {
});
```

This function contains all the unit tests. Create a variable to hold the **AuthenticationService** instance:

```
let authenticationService :
AuthenticationService;
```

Now add a **beforeEach()**:

```
beforeEach(() => {
    authenticationService = new
AuthenticationService();
});
```

All this does is create a new instance of the **authenticationService**.

We only have two conditions to test here, that authentication succeeds, and that authentication fails. Start with a test for successful authentication:

```
it('User Authentication Succeeds', (done: DoneFn)
=> {
});
```

The **done()** function is passed into this test, signifying that it will run asynchronously. Now, call the **authenticate()** and store the Observable to a variable:

```
let o : Observable<ResultObjectVO> =

authenticationService.authenticate("me","me");
```

Subscribe to the **Observable** and put your tests inside it:

```
o.subscribe(value => {
    expect(value.error).toBe(false);
    expect(value.resultObject.userID).toBe(1);

expect(value.resultObject.username).toBe("me");
    expect(value.resultObject.role).toBe(1);
    done();
});
```

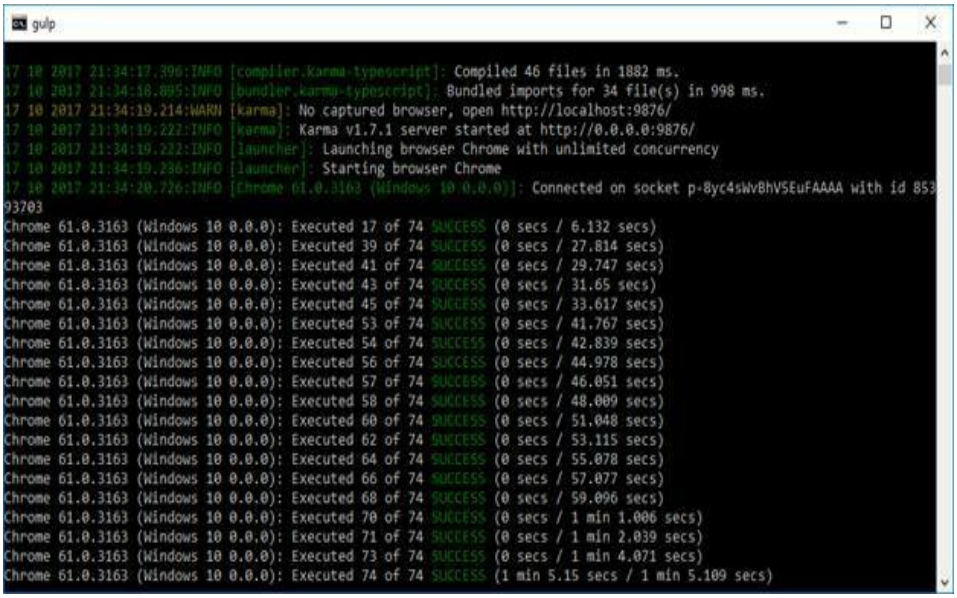
We test that the **error** of the **ResultObjectVO** is “false”. That the **userID** is 1, and the username is “me”. We also check that the user’s **role** is 1. Then the **done()** function is called, telling Jasmine that the unit test is complete.

We use a very similar approach to test failed authentication:

```
it('User Authentication Fails', (done: DoneFn) =>
{
  let o : Observable<ResultObjectVO> =

authenticationService.authenticate("no","no");
  o.subscribe(value => {
      expect(value.error).toBe(true);
      expect(value.resultObject).toBeUndefined();
      done();
  });
});
```

The **Observable** is created by calling the authenticate with credentials that we know will fail. Then we **subscribe** to the results. Inside the **subscribe()** function, we expect the **error** to be true. No user object is returned for a failed login, so we can also check that the **resultObject** is undefined.

Run your tests and you’ll see lots of successful tests:



Review the ColdFusion HttpModule Service

Let's review the Http services that integrate with ColdFusion. Unlike the mock service, the ColdFusion service has dependencies upon the Angular framework. We inject the Angular **Http** service into it and use that as the base of our remote service communication. Open up the **authentication.service.ts** file in the **src/com/dotComIt/learnWith/services/coldFusion** directory. I'm going to show the whole file here. First the imports:

```
import {Injectable} from "@angular/core";
import {Http, RequestOptions, Headers} from
"@angular/http";
import {Observable} from "rxjs/Observable";
import 'rxjs/add/operator/map';
import {Md5} from "ts-md5/dist/md5";
```

We bring in the **Injectable** class so that Angular can inject items into this class when it is created, and to inject other items into this class. A few items are imported from the **@angular/http** library, so we can call the remote service. The **Observable** is brought in to process the results of the service call. The map operator is used to parse the service call results. The **Md5** library is the custom library we use to hash the password before making the service call.

We also need one of the app's custom components:

```
import {ResultObjectVO} from
"../../vo/ResultObjectVO";
```

This is the return object from the service.

The class contains one constant outside of the class:

```
const servicePrefix : string =
'../../coldFusion/';
```

This is used as part of the service call URL, to find the services from the app location to the services location.

Now, create the class:

```
@Injectable()
export class AuthenticationServiceCF {
}
```

The class specifies the injectable metadata so that Angular can use it as a service and also inject services into the class instance. Create a **RequestOptions** instance:

```
options : RequestOptions = new RequestOptions();
```

This is class used to specify the content headers that allow Angular to properly pass data to the server.

Now create the constructor:

```
constructor(private http: Http) {
  let optionHeaders : Headers = new Headers();
  optionHeaders.append('Content-Type',
                       'application/x-www-form-
urlencoded; charset=UTF-8');
  this.options = new
RequestOptions({headers:optionHeaders});
}
```

A content-type header is created and added to the **RequestOptions**. Also, this is where the **Http** service is injected into the class by Angular.

Finally, the **authenticate()** method:

```
authenticate(username :string, password :string)
:Observable<ResultObjectVO>{
  let parameters : string =  "username" + '=' +
username + '&';
  parameters += "password" + '=' +
Md5.hashStr(password) + "&";
  parameters += "method" + "=" + "authenticate" ;
  return this.http.post(servicePrefix +

        'com/dotComIt/learnWith/services/Authenti
                parameters, this.options)
            .map((result) => result.json());
}
```

This is where the magic happens. A parameter string is created. The password is hashed as part of this. Then the **post()** method is called on the **http** service. It specifies the service endpoint, the parameter string, and the options. The **map()** function is run on the results to automatically translate the results from a string into a **JSON** object. That's it. How do we test this method?

Test the ColdFusion HttpModule Service

Create a file called **authentication.service.test.ts** in the **tests/com/dotComIt/learnWith/services/coldFusion** directory. As always, this directory structure parallels the app's source code structure. Let's start with some imports:

```
import {Observable} from "rxjs/Observable";
import {async,TestBed} from
'@angular/core/testing';
import
{HttpModule,Response,XHRBackend,ResponseOptions}
from '@angular/http';
import {MockBackend} from
'@angular/http/testing';
```

A lot of this we've seen before, such as the **Observable** class and the angular testing components. We have some new components

from **@angular/http** that we have not used for testing yet. The **HttpModule** is the module that contains the Http service we use. **Response** and **ResponseOptions** will be used to mock a response from the server. The **XHRBackend** handles the HTTP calls, but we'll replace it with the **MockBackend** from the testing library. We want to test our service calls without actually calling the services.

We need a few of our own imports:

```
import {AuthenticationService} from

"../../../../../../src/com/dotComIt/learnWith/services/r
import {AuthenticationServiceCF} from

"../../../../../../src/com/dotComIt/learnWith/services/coldfusio
import {ResultObjectVO} from

"../../../../../../src/com/dotComIt/learnWith/vo/
import {UserVO} from

"../../../../../../src/com/dotComIt/learnWith/vo/
```

We bring in the mock **AuthenticationService** because that is setup in our sample module in **base.test.ts**. We're going to replace it with the **AuthenticationServiceCF** class. The **ResultObjectVO** and **UserVO** will be used to generate responses from the server.

Create the **describe()** block:

```
describe('ColdFusion Authentication Service',
function () {
});
```

Create a few variables that can be used across multiple tests:

```
let authenticationService :
AuthenticationServiceCF;
let mockBackend : MockBackend;
```

One variable contains the **AuthenticationService**, and one the **MockBackend**.

Now, create a **beforeEach()**.

```
beforeEach(async(() => {
});
```

This before each needs to perform two tasks. The first will reconfigure a few items on the **TestBed ngModule**.

```
TestBed.configureTestingModule({
  providers: [
      { provide: AuthenticationService, useClass:
AuthenticationServiceCF },
      { provide: XHRBackend, useClass:
MockBackend }
  ] ,
  imports : [HttpModule]
});
```

First, it tells Angular that whenever a component calls for the **AuthenticationService** class, to use the **AuthenticationServiceCF** class. Whenever a component calls for the **XHRBackend** class, use the **MockBackend** class. **XHRBackend** is used under the scenes to make the service calls. Finally, it also adds the **HttpModule** to the imports. The reason this config is not set up in the **base.test.ts** is because we only need to perform these replacements to test the ColdFusion services, not the mock services or other components.

The final piece of the **beforeEach()** is to compile the components, and get our component instances:

```
TestBed.compileComponents().then(() => {
    authenticationService =
TestBed.get(AuthenticationService);
    mockBackend = TestBed.get(XHRBackend);
});
```

We're now prepared to write tests for the **authenticate()** method. I'm going to wrap them in another **describe()**:

```
describe('authenticate() ', function () {
});
```

This is so I can create a variable to hold a shared **mockResponse**:

```
let mockResponse : ResultObjectVO;
```

This will variable will be defined in its own **beforeEach()**:

```
beforeEach(() => {
    mockResponse = new ResultObjectVO();
    mockResponse.error = false;
    mockResponse.resultObject = new UserVO();
    mockResponse.resultObject.userID = 1;
    mockResponse.resultObject.username = "me";
    mockResponse.resultObject.role = 1;
});
```

The **mockResponse** is just an instance of the **ResultObjectVO**. This signifies a successful login with the user.

Start the test for successful login:

```
it('User Authentication Succeeds', () => {
});
```

The first thing we're going to do is use the **mockBackend** to create our response:

```
mockBackend.connections.subscribe(
(connection:any ) => {
    connection.mockRespond(new Response(new
ResponseOptions({
        body: JSON.stringify(mockResponse)
    })));
})
```

We drill down into the **connections** property and subscribe. A new **Response** is created. Its argument is a new **ResponseOptions** instance. That contains the body that will be returned as part of the call. The body contains the **mockResponse** object translated into JSON.

Now, call the service:

```
let o : Observable<ResultObjectVO> =
```

```
authenticationService.authenticate("me","me");
```

Subscribe to the results:

```
o.subscribe(value => {
    expect(value.error).toBe(false);
    expect(value.resultObject.userID).toBe(1);

expect(value.resultObject.username).toBe("me");
    expect(value.resultObject.role).toBe(1);
});
```

All of our tests are in the **subscribe()** function. These tests mirror the tests we used for the mock service. They make sure that the **error** value is false, and checks some of the values on the **userVO** in the **resultObject**. That's all that is needed.

Let's write the failure test:

```
it('User Authentication Fails', () => {
});
```

The first thing we need to do is modify the **mockResponse** so the **error** property is true:

```
mockResponse.error = true;
```

Then write the code for the **mockBackend**:

```
mockBackend.connections.subscribe(
(connection:any ) => {
    connection.mockRespond(new Response(new
ResponseOptions({
        body: JSON.stringify(mockResponse)
    })));
})
```

This code is identical to the success code, the main difference being it will return a different **mockResponse** due to our previous changes. Call the service:

```
let o : Observable<ResultObjectVO> =
```

```
authenticationService.authenticate("wrong","wrong
```

And subscribe to the results:

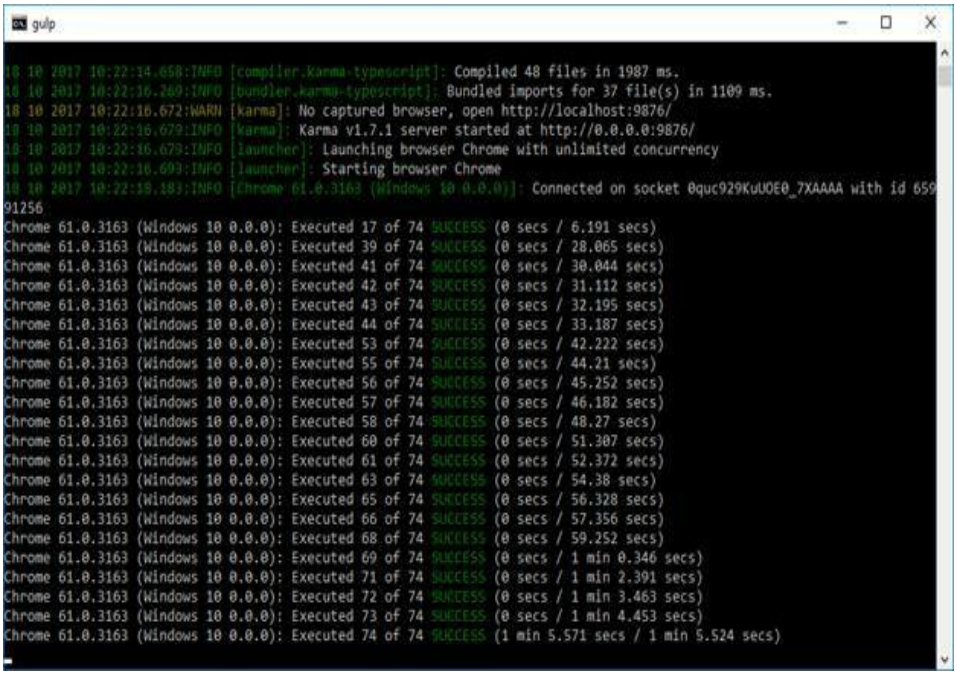
```
o.subscribe(value => {
    expect(value.error).toBe(true);
});
```

This time we only check the assertion to that the error is true. We don't need to drill down into the user object, because with a failed login no user object will be returned.

Let's run the results to see how we're going. Remember to use the specific service we set up for testing ColdFusion code:

```
gulp testColdFusion
```

You'll see results like this:



All good to go.

Review the NodeJS JSONP Service

Let's review the JSONP services that integrate with NodeJS. Unlike

the mock service, the NodeJS service has dependencies upon the Angular framework. We inject the Angular **Jsonp** service into it. Open up the **authentication.service.ts** file in the **src/com/dotComIt/learnWith/services/nodejs** directory. I'm going to show the whole file here. First the imports:

```
import {Injectable} from "@angular/core";
import {Jsonp} from "@angular/http";
import {Observable} from "rxjs/Observable";
import 'rxjs/add/operator/map';
import {Md5} from "ts-md5/dist/md5";
```

We bring in the **Injectable** class so that Angular can inject items into this class when it is created, and to inject other items into this class. The **Jsonp** service is imported from the **@angular/http** library so we can call the remote service. The **Observable** is brought in to process the results of the service call. The map operator is used to parse the service call results. The **Md5** library is the custom library we use to hash the password before making the service call.

We also need one of the app's custom components:

```
import {ResultObjectVO} from
"../../vo/ResultObjectVO";
```

This is the return object from the service.

The file contains one constant outside of the class:

```
const server : string = 'http://127.0.0.1:8080/';
```

This is the URL that will be used to reach the server that contains the service.

Now, create the class:

```
@Injectable()
export class AuthenticationServiceNodeJS {
}
```

The class specifies the injectable metadata so that Angular can use it as a service and also inject services into the class instance.

Now create the constructor:

```
constructor (private jsonp: Jsonp) {
}
```

This just injects the **Jsonp** service. We do not need any custom headers defined like we did when creating the **Http** service.

Finally, the **authenticate()** method:

```
authenticate(username : string, password :
string) :

Observable<ResultObjectVO> {
    let parameters : string =  "username" + '=' +
username + '&';
    parameters += "password" + '=' +
Md5.hashStr(password) + "&";
    parameters += "callback" + "=" +
"JSONP_CALLBACK";
    let url = server + 'login?' + parameters;
    return this.jsonp.request(url)
        .map((result) => {
            return result.json() as
ResultObjectVO;
        });
}
```

This is where the magic happens. A parameter string is created. The password is hashed as part of this. A **callback** argument is added**.** Angular uses this special "JSONP_CALLBACK" value as part of its underlying JSONP implementation. The results will be wrapped in this function, giving Angular access to them.

The full endpoint URL is created, specifying the service, login, and the parameter string. Then the **request()** method is called on the **Jsonp** service. It specifies the service endpoint, the parameter string, and the options. The **map()** function is run on the results to automatically translate the results from a string into a **ResultObjectVO** object. That's it. How do we test this method?

Test the NodeJS JSONP Service

Create a file called **authentication.service.test.ts** in the **tests/com/dotComIt/learnWith/services/nodejs** directory. The tests directory structure parallels the app's source code structure. Let's start with some imports:

```
import {Observable} from "rxjs/Observable";
import {async,TestBed} from
'@angular/core/testing';
import
{JsonpModule,JSONPBackend,Response,ResponseOption
from @angular/http';
import {MockBackend} from
'@angular/http/testing';
```

A lot of this we've seen before, such as the **Observable** class and the angular testing components. We bring some new things from the **@angular/http** library. The **JsonpModule** is the module that contains the **Jsonp** service. **Response** and **ResponseOptions** will be used to mock a response from the server. The **JSONPBackend** handles the service calls, but we'll replace it with the **MockBackend** from the testing library. We want to test our service calls without actually calling the services.

We need a few of our own imports:

```
import {AuthenticationService} from

"../../../../../../src/com/dotComIt/learnWith/services/r
import {AuthenticationServiceNodeJS} from

"../../../../../../src/com/dotComIt/learnWith/services/nodeJS/au
import {ResultObjectVO} from

"../../../../../../src/com/dotComIt/learnWith/vo/
import {UserVO} from

"../../../../../../src/com/dotComIt/learnWith/vo/
```

We bring in the mock **AuthenticationService** because that is setup

in our sample module in **base.test.ts**. We're going to replace it with the **AuthenticationServiceNodeJS** class. The **ResultObjectVO** and **UserVO** will be used to generate responses from the server.

Create the **describe()** block:

```
describe('NodeJS Authentication Service',
function () {
});
```

Create a few variables that can be used across multiple tests:

```
let authenticationService :
AuthenticationServiceNodeJS;
let mockBackend : MockBackend;
```

One variable contains the **AuthenticationService**, and one the **MockBackend**.

Now, create a **beforeEach()**.

```
beforeEach(async(() => {
});
```

This before each needs to perform two tasks. The first will reconfigure a few items on the **TestBed ngModule**.

```
TestBed.configureTestingModule({
  providers: [
   { provide: AuthenticationService, useClass:
AuthenticationServiceNodeJS },
   { provide: JSONPBackend, useClass: MockBackend
}
  ],
  imports : [JsonpModule]
});
```

First, it tells Angular that whenever a component calls for the **AuthenticationService** class, to use the **AuthenticationServiceNodeJS** class. Angular offers this as an easy way to swap out services for each other. Whenever a component calls for the **JSONPBackend** class, we use the **MockBackend** class.

JSONPBackend is used under the scenes to make the service calls. Finally, it also adds the **JsonpModule** to the imports. The reason this config is not set up in the **base.test.ts** is because we only need to perform these replacements to test the ColdFusion services, not the mock services or other components.

The final piece of the **beforeEach()** is to compile the components, and get our component instances:

```
TestBed.compileComponents().then(() => {
    authenticationService =
TestBed.get(AuthenticationService);
    mockBackend = TestBed.get(XHRBackend);
});
```

We're now prepared to write tests for the **authenticate()** method. I'm going to wrap them in another **describe()**:

```
describe('authenticate() ', function () {
});
```

This is so I can create a variable to hold a shared **mockResponse**:

```
let mockResponse : ResultObjectVO;
```

This will variable will be defined in its own **beforeEach()**:

```
beforeEach(() => {
    mockResponse = new ResultObjectVO();
    mockResponse.error = false;
    mockResponse.resultObject = new UserVO();
    mockResponse.resultObject.userID = 1;
    mockResponse.resultObject.username = "me";
    mockResponse.resultObject.role = 1;
});
```

The **mockResponse** is just an instance of the **ResultObjectVO**. This signifies a successful login with the user.

Start the test for successful login:

```
it('User Authentication Succeeds', () => {
});
```

The first thing we're going to do is use the **mockBackend** to create our response:

```
mockBackend.connections.subscribe(
(connection:any ) => {
    connection.mockRespond(new Response(new
ResponseOptions({
        body: JSON.stringify(mockResponse)
    })));
})
```

We drill down into the **connections** property and subscribe. A new **Response** is created. Its argument is a new **ResponseOptions** instance. That contains the body that will be returned as part of the call. The body contains the **mockResponse** object translated into JSON.

Now, call the service:

```
let o : Observable<ResultObjectVO> =

authenticationService.authenticate("me","me");
```

Subscribe to the results:

```
o.subscribe(value => {
    expect(value.error).toBe(false);
    expect(value.resultObject.userID).toBe(1);

expect(value.resultObject.username).toBe("me");
    expect(value.resultObject.role).toBe(1);
});
```

All our tests our in the subscribe function. These tests mirror the tests we used for the mock service. They make sure that the **error** value is false, and checks some of the values on the **userVO** in the **resultObject**. That's all that is needed.

Let's write the failure test:

```
it('User Authentication Fails', () => {
});
```

The first thing we need to do is modify the **mockResponse** so the **error** property is true:

```
mockResponse.error = true;
```

Then write the code for the **mockBackend**:

```
mockBackend.connections.subscribe(
(connection:any ) => {
    connection.mockRespond(new Response(new
ResponseOptions({
        body: JSON.stringify(mockResponse)
    })));
})
```

This code is identical to the success code, the main difference being it will return a different **mockResponse** due to our previous changes. Call the service:

```
let o : Observable<ResultObjectVO> =

authenticationService.authenticate("wrong","wrong
```

And subscribe to the results:

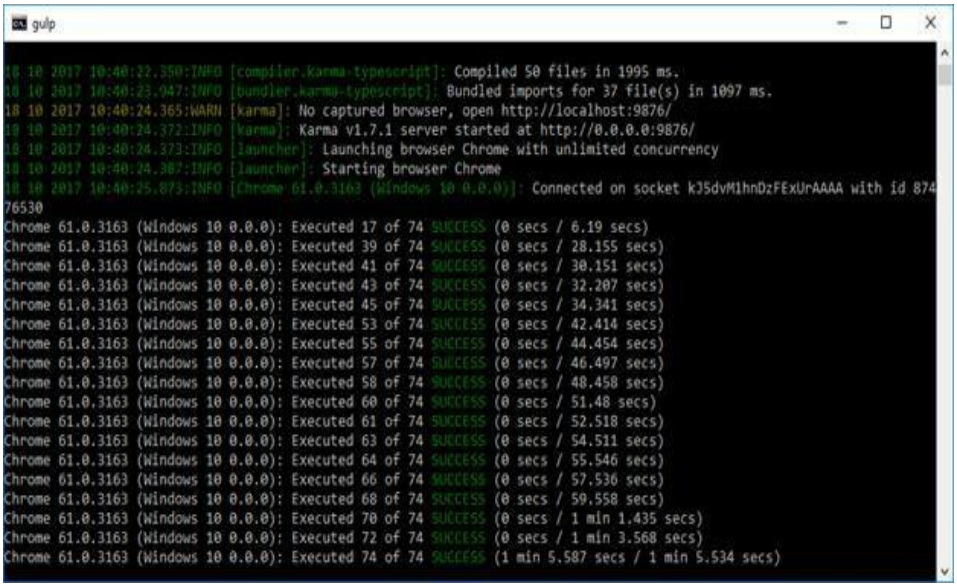
```
o.subscribe(value => {
    expect(value.error).toBe(true);
});
```

This time we only check the assertion to that the error is true. We don't need to drill down into the user object, because with a failed login no user object will be returned.

Let's run the results to see how we're going. Remember to use the specific service we set up for testing NodeJS code:

```
gulp testNodeJS
```

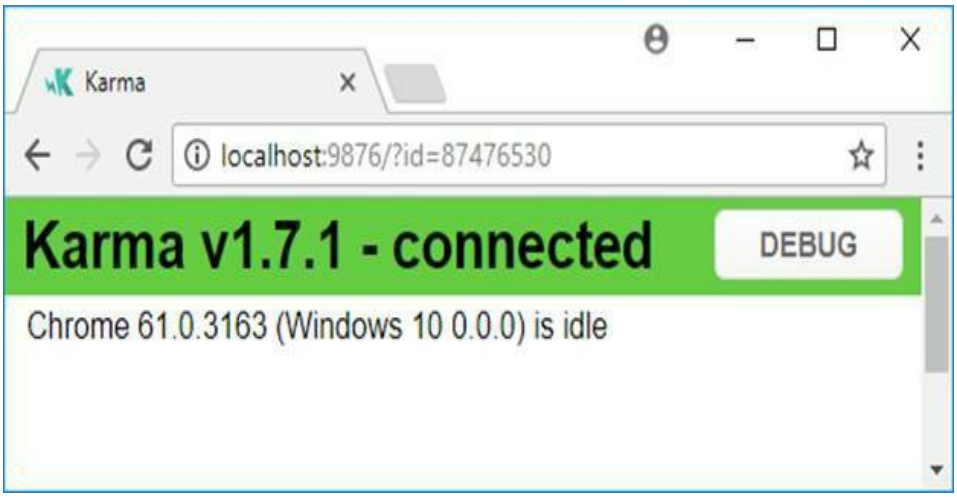
You'll see results like this:



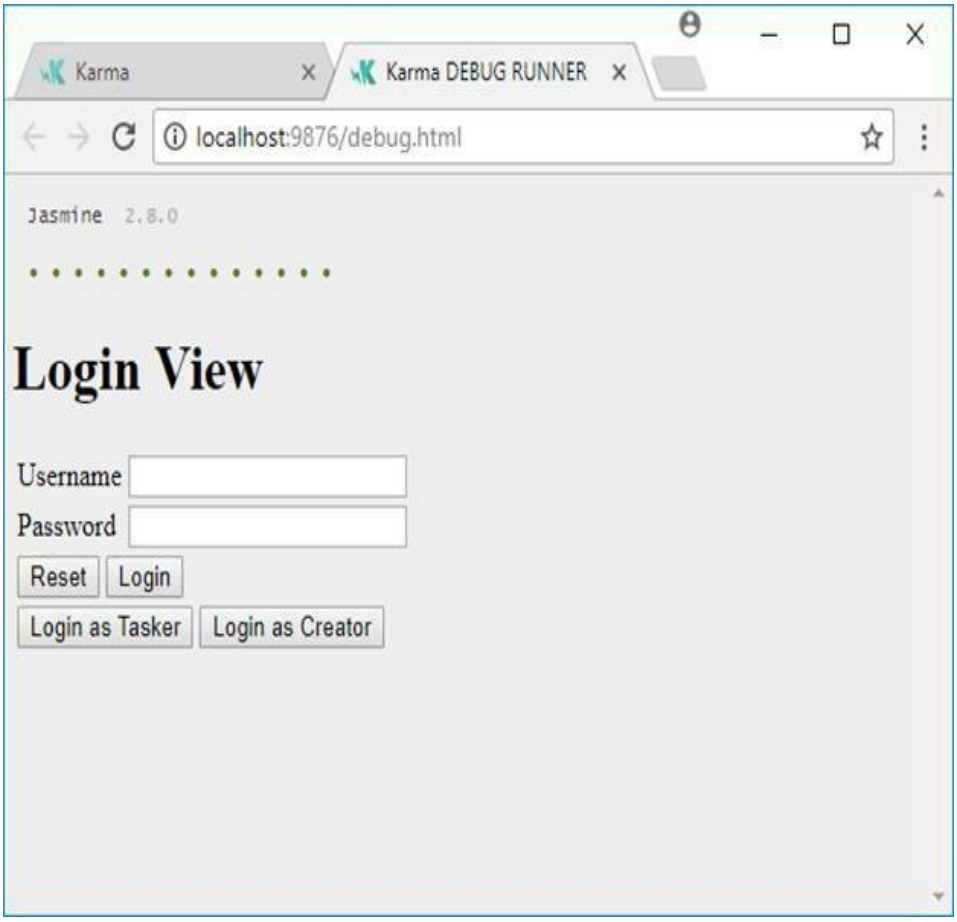
Test are all clear.

Debug Your Tests

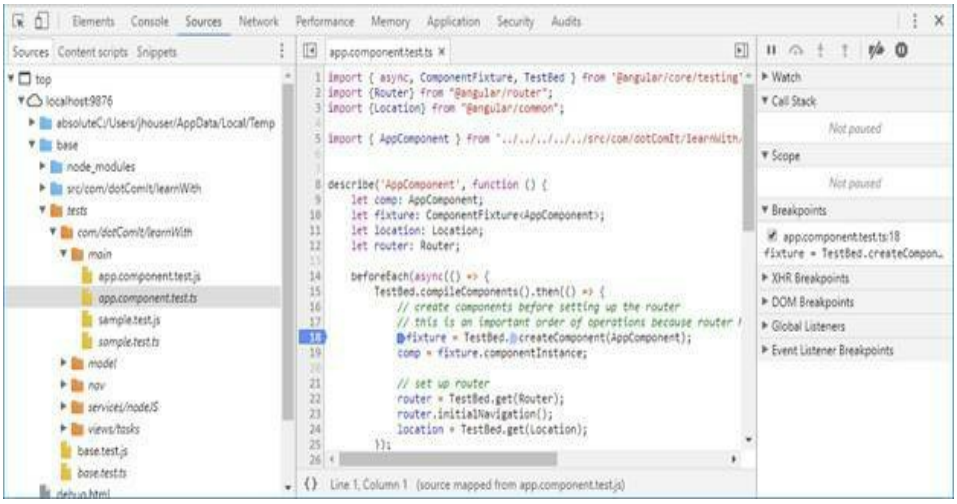
An interesting aspect of using the chrome browser to run your tests is that you can debug the tests. Look at the browser when you're running tests:



Click the debug button, you'll see something like this:



A new tab opens. You'll see dots up top as each test and the HTML template is rendered to the browser as test are run. But, the most important part is you can bring up Chrome's Developer Tools:



You can view the tests or the original code in the sources tab. You can see errors in the console, if they exist. With the current code, you probably noticed a bunch of these errors:

Because the web server is not properly remapping the **calendar-icon.svg** so it can't find it. It doesn't affect the unit testing despite being a minor annoyance.

You can add breakpoints to stop the execution and step through tests just like you were debugging the real app. Sometimes this can be very useful to determine if you're having a problem writing your test, or have discovered a bug in the underlying code. I wrote more about debugging JavaScript elsewhere in this book, and you can use most of the same tactics to debug your tests this way.

Hide console.log( ) with Karma Config

You'll notice there is some log output to the screen when running tests. I tried to minimize that in the past screenshots as much as possible, but a few places it comes through. This comes from **console.log()** statements I put inside the controller during development. You can silence this output using the **client.captureConsole** console property in the **Karma.conf.js** file:

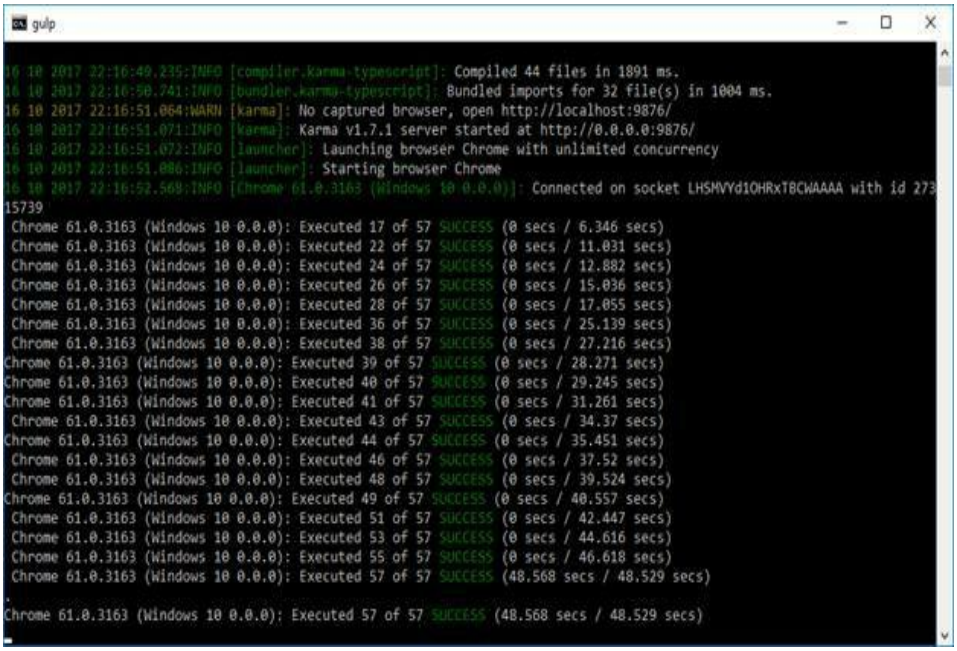
```
client : {
    captureConsole : false
},
```

I like to see the output from the test code when writing and debugging unit tests; but do not like to see the output from the app code. There isn't an easy way to turn it on for one and off for another.

While we're at it, you'll notice that a line is output for every test. We can simplify that by specifying a **Karma** reporter. In the **karma.config.js** file, add the dots reporter:

```
reporters: ["progress", "kjhtml", "dots"]
```

The dots reporter is built into Karma. With these two modifications, rerun the tests:



These properties provide us with cleaner output that is more easily parsable if we have unit test errors.

Final Thoughts

At this point you should have a good understanding of unit tests—what they are, and what it takes to write unit tests against Angular code. You can find all the code for the test and scripts in the source repository from the www.learn-with.com site.

Afterword

I wrote this book to document my own learning process building HTML5 applications. The main series focuses on building an application from start to finish. This book focuses on all the surrounding technology you'll probably need to know on the job. I hope you benefited from my experience.

If you want more information, then be sure to check out **www.learn-with.com**. You can test the application, get the source code for each chapter, get the most up-to-date version of the books, and browse some of our other titles, which will build the same app using different technologies

If you need personal mentoring or have a custom consulting project, we'd love to help out, so please reach out.

Send me an email at jeffry@dot-com-it.com and tell me how this book helped you become a success.