.NET Framework: Developing Modern Web Apps with ASP.NET MVC – Workshop*PLUS*

Module 6: Client-side Development

Student Lab Manual

Instructor Edition (Book Title Hidden Style)

Version 1.0

**Conditions and Terms of Use**

**Microsoft Confidential**

This training package is proprietary and confidential, and is intended only for uses described in the training materials. Content and software is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or disclosing all or any portion of the content and/or software included in such packages is strictly prohibited.

The contents of this package are for informational and training purposes only and are provided "as is" without warranty of any kind, whether express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Training package content, including URLs and other Internet Web site references, is subject to change without notice. Because Microsoft must respond to changing market conditions, the content should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

© 2017 Microsoft Corporation. All rights reserved.

**Copyright and Trademarks**

© 2017 Microsoft Corporation. All rights reserved.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

For more information, see Use of Microsoft Copyrighted Content at  
[*http*://www.microsoft.com/about/legal/permissions/](http://www.microsoft.com/about/legal/permissions/)

IntelliSense, Internet Explorer, Microsoft, Visual Studio, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other Microsoft products mentioned herein may be either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are property of their respective owners.

Contents

[Lab 6: Client-side Development 1](#_Toc480802924)

[Exercise 1: Using Grunt, NPM and Visual Studio Task Runner 4](#_Toc480802925)

[Exercise 2: Using Gulp 13](#_Toc480802926)

[Exercise 3: Using Bower 16](#_Toc480802927)

[Exercise 4: Using Angular 2 21](#_Toc480802928)

# Lab 6: Client-side Development

#### Introduction

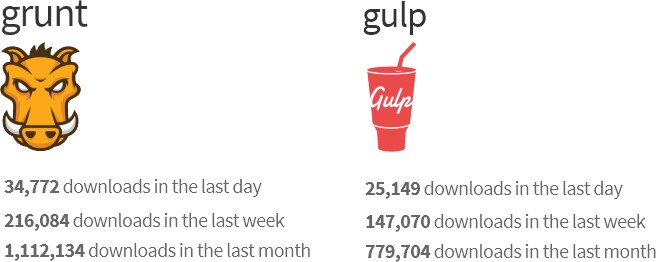
Both Grunt and Gulp are JavaScript task runners that automate script minification, TypeScript compilation, code quality “lint” tools, CSS pre-processors, and just about any repetitive chore that needs doing to support client development.

**What’s the difference between Grunt and Gulp?**

Grunt is an early entry in the client-build-tool space. Grunt modules predefine most everyday tasks like linting, minimizing, and unit testing. Grunt is widely adopted and downloaded thousands of times each day.

While Gulp is a later entry to the field, Gulp has gained popularity for crisp performance and elegant syntax. While Grunt tends to read and write files to disk, Gulp uses a stream ([Vinyl)](https://www.npmjs.com/package/vinyl) object to pipe method results to following methods, allowing calls to be chained together in a fluent syntax.

The statistics below is a screenshot from the [npmjs](https://www.npmjs.com/) (node package manager) home site downloads.



**Bower**

[Bower](http://bower.io/) is a package manager that delivers groups of files used at client-side at run-time. For example, with Bower you can install CSS, fonts, frameworks, and JavaScript libraries from external sources. Bower resolves dependencies and will automatically download and install all the packages you need. For example, if you configure Bower to load the Bootstrap package, the proper jQuery package will automatically come along for the ride.

Visual Studio developers are already familiar with the [NuGet](https://www.nuget.org/) package manager, so why not use NuGet instead of adding yet another tool? Mainly because Bower already has a rich eco-system with [about 18 thousand packages in play](http://bower.io/stats/) and because ASP.NET is no longer strictly a Windows and Visual Studio space. Bower is accessible within Visual Studio, but also from the command line and Integrated Development Environments (IDEs) for other environments.

You could also use Node Package Manager ([NPM](https://www.npmjs.com/)), but NPM typically loads client development, debugging, and testing tools like Less, [JSLint](http://www.jslint.com/)[, QUnit,](http://qunitjs.com/) and [Grunt.](http://gruntjs.com/) While NPM uses a nested dependency tree, Bower has a flatter structure and for that reason tends to be lighter weight. For example, in NPM, you can easily have different versions of the same component while Bower does not allow two components with the same name.

*“Bower is optimized for the front-end. Bower uses a flat dependency tree, requiring only one version for each package, reducing page load to a minimum.”*

*– Excerpt from the* [*http://bower.io*](http://bower.io/) *main page*

**How does Bower work – the short story**

Bower works together with the NPM and Grunt (a client-side task runner). NPM first loads Grunt and a Grunt task that supports Bower. Grunt then installs Bower. Finally, Bower resolves dependencies, downloads packages, and deploys the packages to the root of your web site.

Bower packages are essentially Git repositories. Each package has a **bower.json** file that describes the package, such as the name, version, and dependencies. You can add a bower.json file to your own application that defines the packages, your application needs.

#### Objectives

This lab will show you how to:

* Use Grunt and Gulp, JavaScript, and task runners
* Use Visual Studio Task Runner
* Use NPM, and bower package managers for client-side
* AngularJS with ASP.NET Model-View-Controller (MVC)

#### Prerequisites (if applicable)

This lab does not build on previous labs and neither is its end-solution used in the following labs.

#### System Requirements

To complete this lab, you need:

* Visual Studio 2017

#### Hosted Lab Credentials

If the lab is exercised in Microsoft cloud environment, use the following user credentials to sign in:

* Username: aspnetuser
* Password: @Cir9hvc6!w

#### Estimated Time to Complete This Lab

90 minutes

## Exercise 1: Using Grunt, NPM and Visual Studio Task Runner

#### Objectives

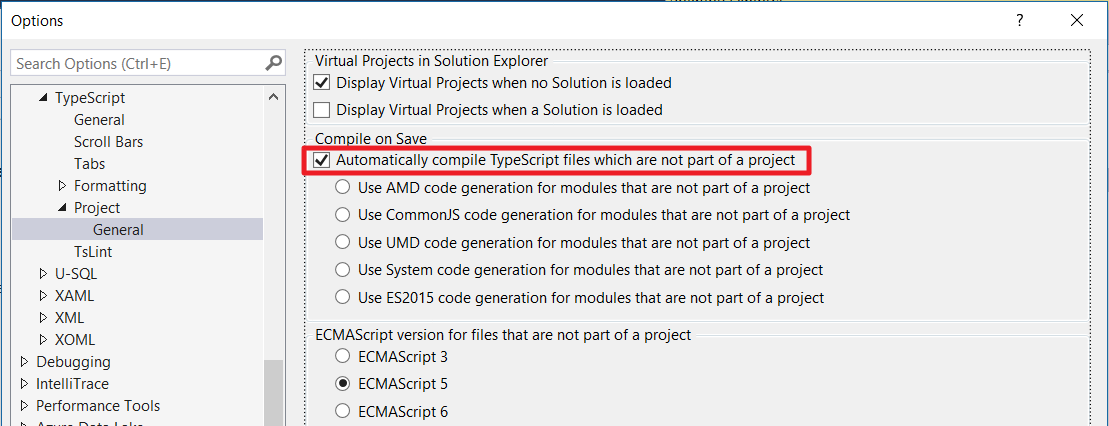
In this exercise, you will:

* Use Grunt (JavaScript task runner) using Visual Studio Task Runner to automate script minification, TypeScript compilation, code quality “lint” tools, CSS pre-processors, and just about any repetitive chore that needs to support client development.

Task 1: Preparing the application

To begin, set up a new empty web application and add TypeScript example files. TypeScript files are automatically compiled into JavaScript using default Visual Studio 2017 settings and will be our raw material to process using Grunt.

1. In Visual Studio 2017, create a new **ASP.NET Web Application**.
2. In the **New ASP.NET Web Application** dialog box, select the **Empty** template and click **OK**.
3. Add a new folder named **TypeScript** to your project directory.
4. Before adding any file, make sure that Visual Studio 2017 has the option **compile on save** for TypeScript files selected. Go to ***Tools*** *>* ***Options*** *>* ***Text Editor*** *>* ***Typescript*** *>* ***Project***



1. Right-click the **TypeScript** directory and select **Add > New** Item from the context menu. Select the **TypeScript file** item and name the file **Tastes.ts** (note the \*.ts extension). Copy the line of TypeScript code below into the file (when you save, a new Tastes.js file will appear with the JavaScript source).

enum Tastes { Sweet, Sour, Salty, Bitter }

1. Add a second file using the **TypeScript File** template to the **TypeScript** directory and name it **Food.ts**. Copy the code below into the file.

class Food {

constructor(name: string, calories: number) {

this.\_name = name;

this.\_calories = calories;

}

private \_name: string;

get Name() {

return this.\_name;

}

private \_calories: number;

get Calories() {

return this.\_calories;

}

private \_taste: Tastes;

get Taste(): Tastes {

return this.\_taste;

}

set Taste(value: Tastes) {

this.\_taste = value;

}

}

Task 2: Configuring NPM

Next, configure NPM to download *grunt* and *grunt-tasks*.

1. In the Solution Explorer, right-click the project and select **Add > New Item** from the context menu. Select the **NPM configuration file** item, leave the default name, **package.json**, and click the **Add** button.
2. In the **package.json** file, inside the **devDependencies** object braces, enter “grunt”. Select **grunt** from the IntelliSense list and press **Enter**. Visual Studio will quote the grunt package name, and add a colon. To the right of the colon, select the latest stable version of the package from the upper section of the IntelliSense list (press **Ctrl-Space** if IntelliSense does not appear).

**Note**: NPM uses [semantic versioning](http://semver.org/) to organize dependencies. Semantic versioning, also known as *SemVer*, identifies packages with the numbering scheme **<major>.<minor>.<patch>**. IntelliSense simplifies semantic versioning by showing only a few common choices. The top item in the IntelliSense list (**0.4.5** in the example above) is considered the latest stable version of the package. The carat ^ symbol matches the most recent major version and the tilde ~ matches the most recent minor version. See the [NPM SemVer version parser reference](https://www.npmjs.com/package/semver) as a guide to the full expressivity that SemVer provides.

1. Add more dependencies to load grunt-contrib\* packages for *clean*, *jshint*, *concat*, *uglify* and *watch* as shown in the example below. The versions do not need to match the example.

"devDependencies": {

"grunt": "^1.0.1",

"grunt-contrib-clean": "^1.0.0",

"grunt-contrib-jshint": "^1.1.0",

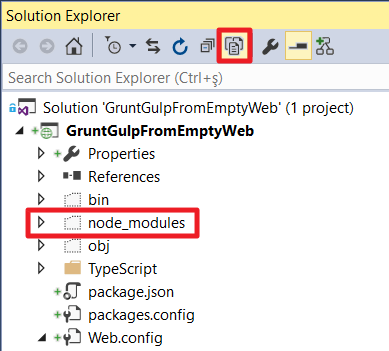
"grunt-contrib-concat": "^1.0.1",

"grunt-contrib-uglify": "^2.0.0",

"grunt-contrib-watch": "^1.0.0"

}

Save the **packages.json** file. Visual Studio will automatically restore packages. Npm packages are restored locally in **npm\_modules** folder. Click **Show All Files** on the toolbar to view where packages are installed.



**Note:** An npm package needs to be installed locally if you are going to **require()** it. If it is going to be used in command line, it should be installed globally.

For more information on folder structures required by npm, see:

<https://docs.npmjs.com/files/folders>

Task 3: Configuring Grunt

Grunt is configured using a manifest named **gruntfile.js** that defines, loads and registers tasks that can be run manually or configured to run automatically based on events in Visual Studio.

1. Right-click the project and select **Add > New Item**. Select the **Grunt Configuration file** option, leave the default name, **gruntfile.js**, and click the **Add** button.

The initial code includes a module definition and the **grunt.initConfig()** method. The initConfig() method is used to set options for each package, and the remainder of the module will load and register tasks.

module.exports = function (grunt) {

grunt.initConfig({ });

};

1. Inside the **initConfig()** method, add options for the **clean** task as shown in the following example, **Gruntfile.js**. The clean task accepts an array of directory strings. This task removes files from **Script/app** and removes the entire **/temp** directory.

module.exports = function (grunt) {

grunt.initConfig({

clean: ["Scripts/app/\*", "temp/"]

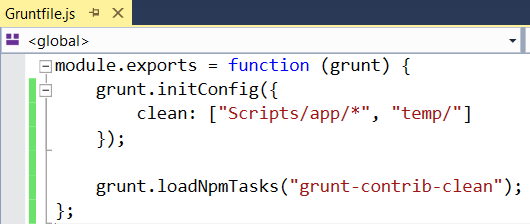
});

};

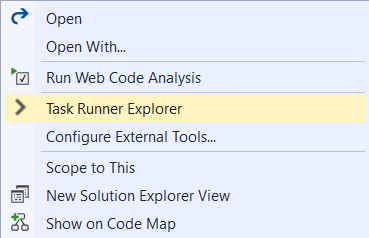
1. Below the **initConfig()** method, add a call to **grunt.loadNpmTasks()**. This will make the task runnable from Visual Studio.

grunt.loadNpmTasks("grunt-contrib-clean");

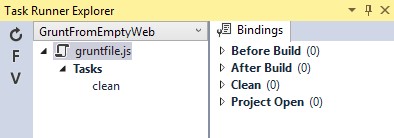
1. Save **Gruntfile.js**. The file should look something like the following screenshot:.



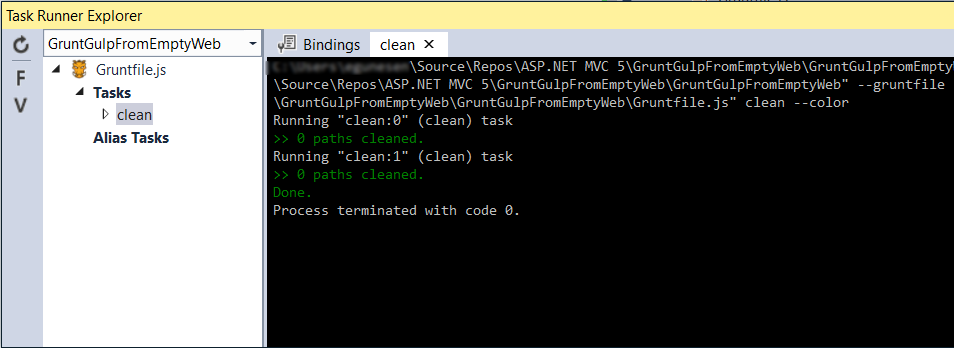
1. Right-click **Gruntfile.js** and select **Task Runner Explorer** from the context menu. The Task Runner Explorer window will open.



1. Verify that **clean** shows under **Tasks** in the Task Runner Explorer. Note that you may need to click the **Refresh** button (upper-left).



1. Right-click the **clean** task and select **Run** from the context menu. A command window displays progress of the task.



**Note:** There are no files or directories to clean yet. If you like, you can manually create them in the **Solution Explorer** and then run the clean task as a test.

1. In the **initConfig()** method, add an entry for **concat** using the code below.

The **src** property array lists files to combine, in the order that they should be combined. The **dest** property assigns the path to the combined file that is produced.

concat: {

all: {

src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],

dest: 'temp/combined.js'

}

},

**Note:** The **all** property in the code above is the name of a *target*. Targets are used in some Grunt tasks to allow multiple build environments. You can view the built-in targets using IntelliSense or assign your own.

1. Add the **jshint** task using the code below.

The jshint code-quality utility is run against every JavaScript file found in the temp directory.

jshint: {

files: ['temp/\*.js'],

options: {

'-W069': false

}

},

**Note**: The option “-W069” is an error produced by jshint when JavaScript uses bracket syntax to assign a property instead of dot notation, that is **Tastes[“Sweet”]** instead of **Tastes.Sweet**. The option turns off the warning to allow the rest of the process to continue.

1. Add the **uglify** task using the code below.

The task minifies the **combined.js** file found in the temp directory and creates the result file in **Scripts/app** following the standard naming convention *<file name>.min.js*.

uglify: {

all: {

src: ['temp/combined.js'],

dest: 'Scripts/app/combined.min.js'

}

}

1. Under the call **grunt.loadNpmTasks()** that loads grunt-contrib-clean, include the same call for jshint, concat and uglify using the code below.

grunt.loadNpmTasks('grunt-contrib-jshint');

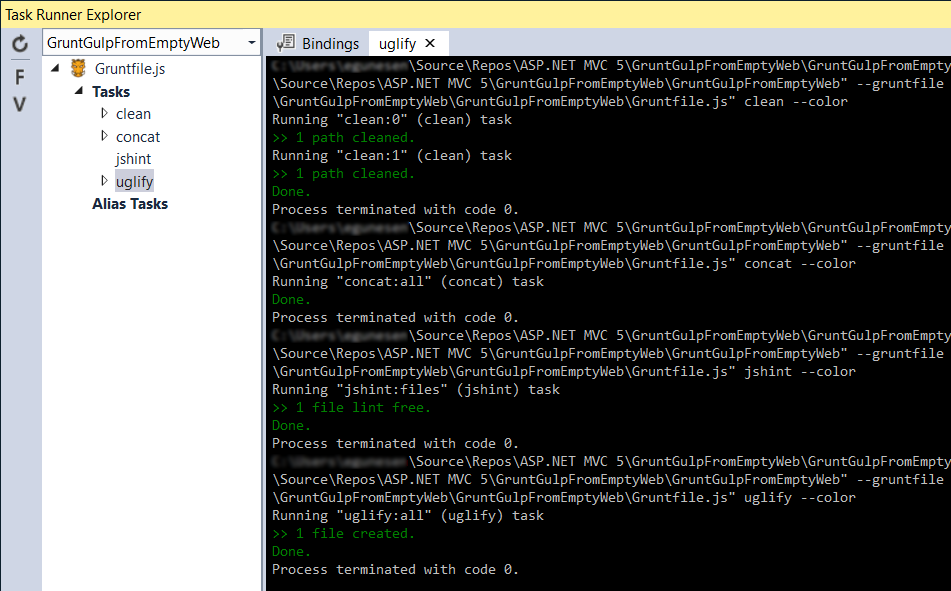
grunt.loadNpmTasks('grunt-contrib-concat');

grunt.loadNpmTasks('grunt-contrib-uglify');

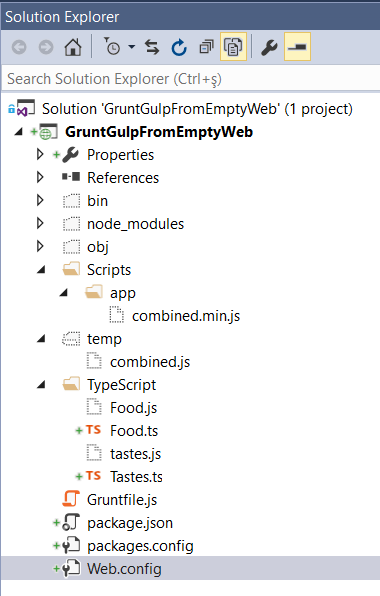
1. Save **Gruntfile.js**. The file should look similar to the example in the following screenshot:



1. Notice that the Task Runner Explorer Tasks list includes **clean**, **concat**, **jshint** and **uglify** tasks. Run each task in order and observe the results in Solution Explorer. Each task should run without errors.



The concat task creates a new **combined.js** file and places it into the **temp** directory. The jshint task simply runs and does not produce the output. The uglify task creates a new **combined.min.js** file and places it into **Scripts/app**. On completion, the solution should look similar to the following screenshot:



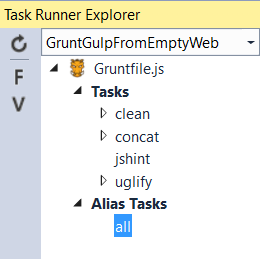
**Note:** For more information on the options for each package, see: <https://www.npmjs.com/>and lookup the package name in the search box on the main page. For example, you can look up the **grunt-contrib-clean** package to get a documentation link that explains all the parameters.

Task 4: All together now

1. Use the Grunt **registerTask()** method to run a series of tasks in a particular sequence. For example, to run the example steps above in the order clean > concat > jshint > uglify, add the code below to the module. The code should be added to the same level as the loadNpmTasks() calls, outside initConfig.

grunt.registerTask("all", ['clean', 'concat', 'jshint', 'uglify']);

1. The new task shows up in Task Runner Explorer under Alias Tasks. You can right-click and run it just as you would run other tasks. The **all** task will run **clean**, **concat**, **jshint** and **uglify**, in order.



Task 5: Watching for changes

1. A **watch** task keeps an eye on files and directories. The watch triggers tasks automatically if it detects changes. Add the code below to initConfig to watch for changes to \*.js files in the **TypeScript** directory. If a JavaScript file is changed, **watch** will run the **all** task.

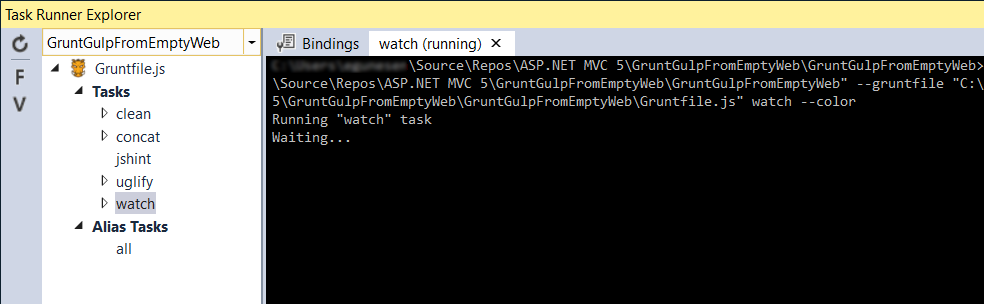
watch: {

files: ["TypeScript/\*\*/\*.js"],

tasks: ["all"]

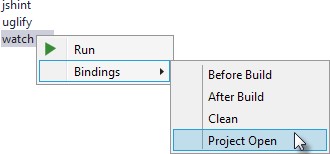
}

1. Add a call to **loadNpmTasks()** to show the **watch** task in Task Runner Explorer. grunt.loadNpmTasks('grunt-contrib-watch');
2. Right-click the **watch** task in Task Runner Explorer and select **Run** from the context menu. The command window that shows the watch task running will display a **waiting** message. Open one of the TypeScript files, make a change that is not just whitespace, and then save the file. This will trigger the **watch** task and trigger the other tasks to run in order. The following screenshot shows a sample run.



Task 6: Binding to Visual Studio Events

1. Unless you want to manually start your tasks every time you work in Visual Studio, you can bind tasks to **Before Build**, **After Build**, **Clean**, and **Project Open** events.
2. Bind **watch** so that it runs every time Visual Studio opens. In **Task Runner Explorer**, right-click the watch task and select **Bindings > Project Open** from the context menu.



1. Unload and reload the project. When the project loads again, the watch task will start running automatically.

Exercise 2: Using Gulp

#### Objectives

In this exercise, you will:

* Use Gulp (JavaScript task runner) using Visual Studio Task Runner to automate tasks that were automated by Grunt in the previous exercise.

Task 1: Configuring NPM

Gulp configuration is similar to Grunt with some notable differences. The example below parallels the Grunt example using Gulp packages and conventions.

* The **devDependencies** defined in **package.json** are specific to Gulp. To get the same result as the Grunt walk-through, **package.json** should add the following gulp packages. You can get the correct version number using IntelliSense (Ctrl + space).

"gulp": "^3.9.1",

"gulp-clean": "^0.3.2",

"gulp-jshint": "^2.0.4",

"gulp-concat": "^2.6.1",

"gulp-uglify": "^2.0.0",

"gulp-rename": "^1.2.2"

Task 2: Configuring Gulp

1. Instead of adding **Gruntfile.js** to the project, add a **Gulp Configuration** **File** to the project and name it **gulpfile.js**. In **gulpfile.js,** assign a series of objects using the **node.js** **require()** method. Make the assignment for Gulp itself and for every package needed for automation. The code below assigns the same tasks used in the Grunt example:

var gulp = require('gulp');

var clean = require('gulp-clean');

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

var jshint = require('gulp-jshint');

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

var rename = require('gulp-rename');

1. Below these assignments in gulpfile.js, call the **gulp** object **task()** method. The first parameter to task() is the name of the task and the second is a function.

gulp.task("all", function () {

});

1. Add the empty task() method to gulpfile.js and it will display the **all** task in Task Runner Explorer.



1. Inside the **task()** function, use the objects defined earlier by **require()** method to do the work. The example below cleans any files from the **Scripts/app** directory.

gulp.task("all", function () {

gulp.src('Scripts/app/\*').pipe(clean());

});

Task 3: All together

Gulp is a streaming object that includes methods **src(),** **pipe(),** and **dest()**.

* **src()** defines where the stream is coming from -- **Scripts/app** in our example. The method returns a stream that can be passed to other Gulp plugins.
* **pipe()** pulls data from the stream and writes it to the destination parameter.
* **dest()** outputs streams to files.

The general coding pattern for Gulp looks like this partial example:

gulp.src()

.pipe()

.pipe()

.pipe(dest());

The src() method gathers the initial raw materials. A series of pipe() calls allow Gulp plugins to operate on the stream. Finally, the dest() method writes out the final results. The advantage to this flow is that only one file read and one file write occur, making the whole process quicker.

1. Here is the complete code that concatenates, lints, minifies and writes the minified file. The processing time is quite fast.

gulp.task("all", function () {

gulp.src('Scripts/app/\*').pipe(clean());

gulp.src(['TypeScript/Tastes.js', 'TypeScript/Food.js'])

.pipe(concat("combined.js"))

.pipe(jshint())

.pipe(uglify())

.pipe(rename({

extname: '.min.js'

}))

.pipe(gulp.dest('Scripts/app'))

});

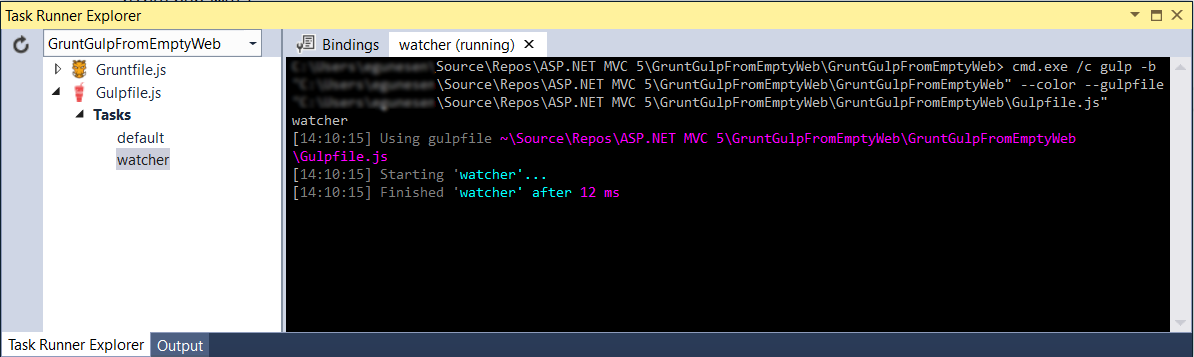
1. Watcher tasks are similar to the Grunt parallel task and are simple to set up. Again, the gulp.task() method names the task that will show in the Task Runner Explorer. The Gulp **watch()** method takes a path or array of paths and second parameter is an array of tasks to run.

gulp.task("watcher", function () {

gulp.watch("TypeScript/\*.js", ['all']);

});

1. The Task Runner Explorer running Gulp tasks uses the same interface as Grunt. The following screenshot shows the **watcher** task running.



**Summary**

Both Grunt and Gulp are powerful tasks runners that automate most client-build tasks. Grunt and Gulp both require support from NPM to deliver their packages. While Grunt is configured using Gruntfile.js and Gulp is configured using Gulpfile.js, both build tools play nicely in Visual Studio, automatically sensing changes to the configuration files. Task Runner Explorer detects changes to configuration files and provides a convenient interface to run tasks, view running tasks, and bind tasks to Visual Studio events.

Exercise 3: Using Bower

#### Objectives

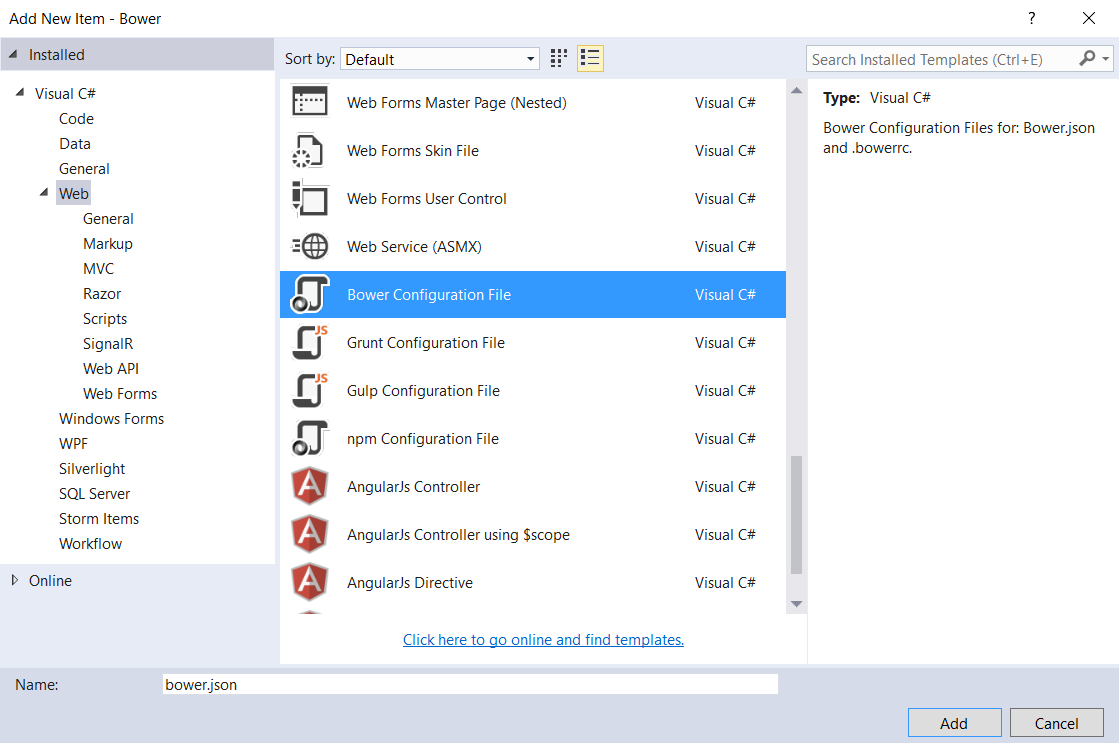
In this exercise, you will:

* Use Bower (package manager for the web) to add a client-side package and use it in HTML page.

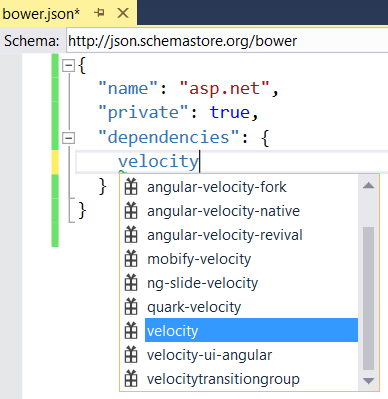
Task 1: Configure and use Bower Package

In Visual Studio, Bower Package Manager can be used to add packages to a project. Let us create an ASP.NET MVC Web Application and add **velocity** package to add animation support with Bower.

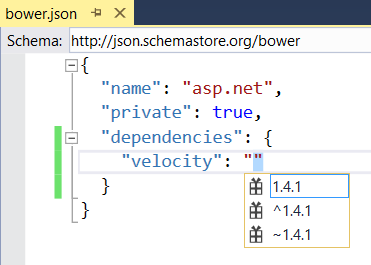
1. In Visual Studio 2017, create a new **ASP.NET Web Application**.
2. In the **New ASP.NET Web Application** dialog box, select the **MVC** template and click **OK**.
3. Right-click to the web application, click **Add > New Item**, and then select Bower Configuration File.



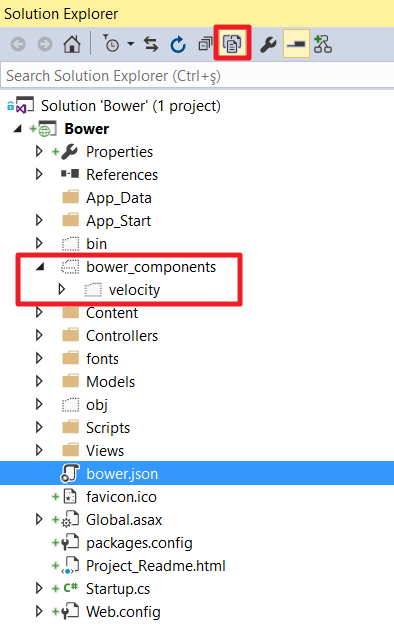
1. At the end of the **dependencies** properties listing in **bower.json** file, add a comma and type *velocity*. Select **velocity** from the drop-down list.



1. Add a colon and then select the latest stable version of the package from the drop-down list.



1. Save the **bower.json** file.
2. In the Solution Explorer, right-click the **bower.json** file and select **Restore Packages** from the drop-down menu.
3. In **Solution Explorer** toolbar, click **Show All Files** button. You will see **bower\_components** folder containing velocity package.



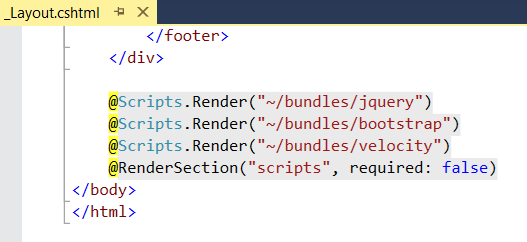
1. To add reference to velocity package in the web page, open **BundleConfig.cs** and add the following line:

bundles.Add(new ScriptBundle("~/bundles/velocity").Include( "~/bower\_components/velocity/velocity.min.js"));

1. To make sure the velocity package is available for all views, open **\_Layout.cshtml** and add the following line after jquery bundle reference.

@Scripts.Render("~/bundles/velocity")

The page should now look similar to the following:



**Note**: Since velocity package depend on jquery, it should be placed after jquery bundles.

1. Now everything is ready, you can use velocity to give animations to div elements when hovering on them. Open **Index.cshtml** and add the following lines at the end of the file:

@section scripts {

<script type="text/javascript">

$(document).ready(function () {

$('div.row div').hover(

function () { },

function () { }

);

});

</script>

}

This script will find all div elements under the div having css class *row*. It will add one function which will be run when mouse pointer hovers on the div, and a second one which will be run when mouse pointer leaves the div.

1. Add velocity codes into the first function to make the div element bigger in 300 milliseconds. Then add code to the second function to revert the element back to its normal size.

The script should now look similar to the following:

@section scripts {

<script type="text/javascript">

$(document).ready(function () {

$('div.row div').hover(

function () {

$(this)

.velocity({ scale: 1.05 }, { duration: 300 })

.toggleClass("popped");

},

function () {

$(this)

.velocity({ scale: 1 }, { duration: 300 })

.toggleClass("popped");

}

);

});

</script>

}

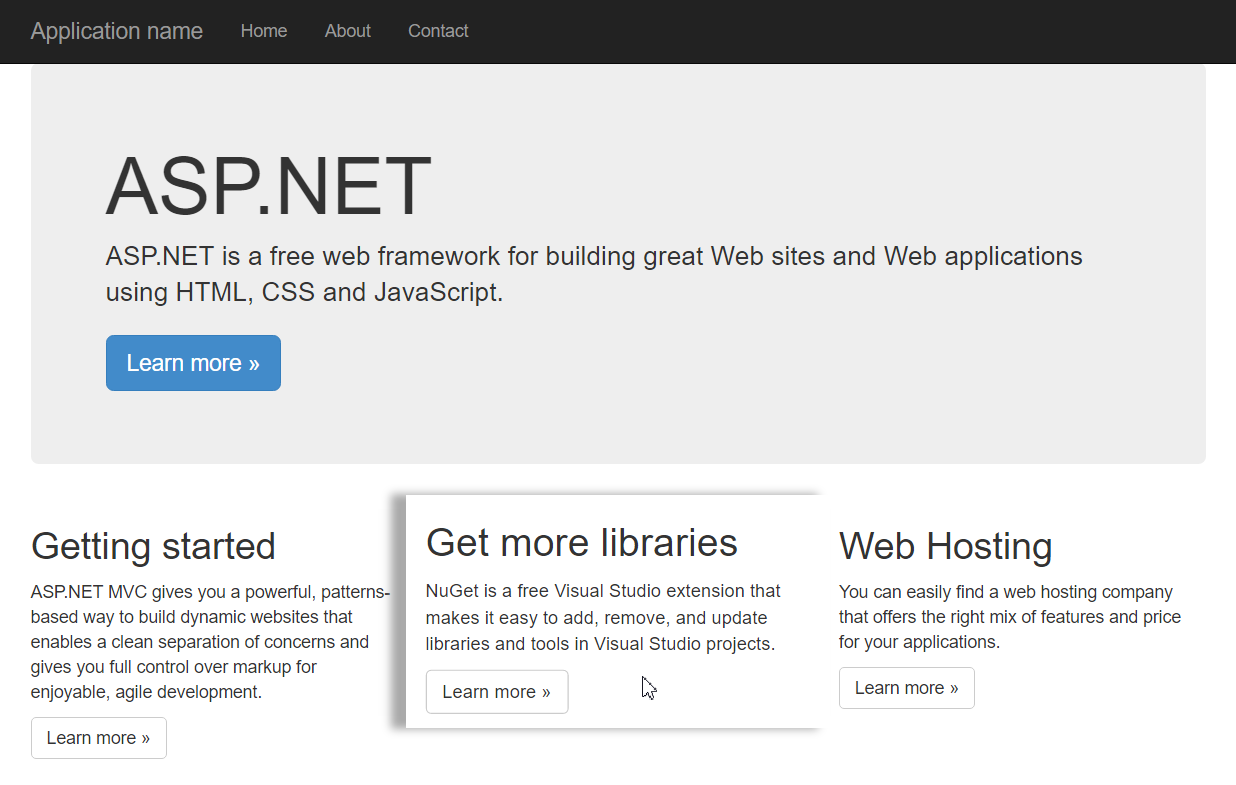
1. The previous script also adds css class **popped** to div elements, with **toggleClass** jquery method. To make this css class give shadow effect, open **Site.css** and add the following line:

div.row div.popped {

box-shadow: -10px 0px 10px 1px #aaaaaa;

}

1. Press **F5** to run the project. On the home page, hover on one of the three sections at the bottom and see it slowly popping up.



Exercise 4: Using Angular 2

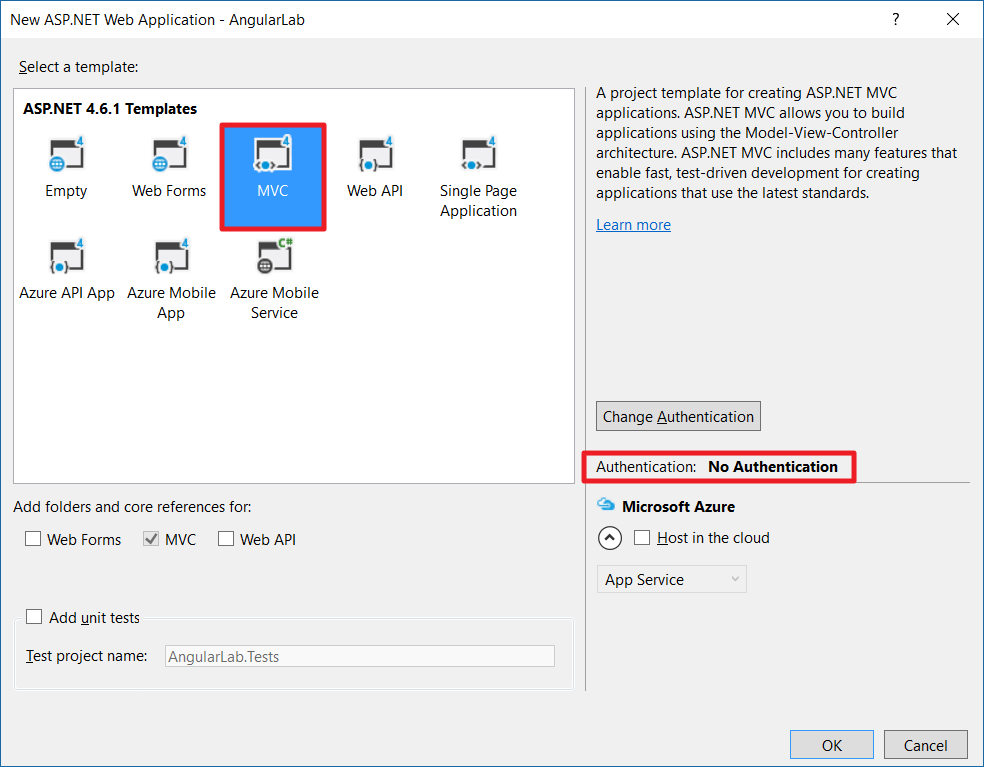
#### Objectives

In this exercise, you will:

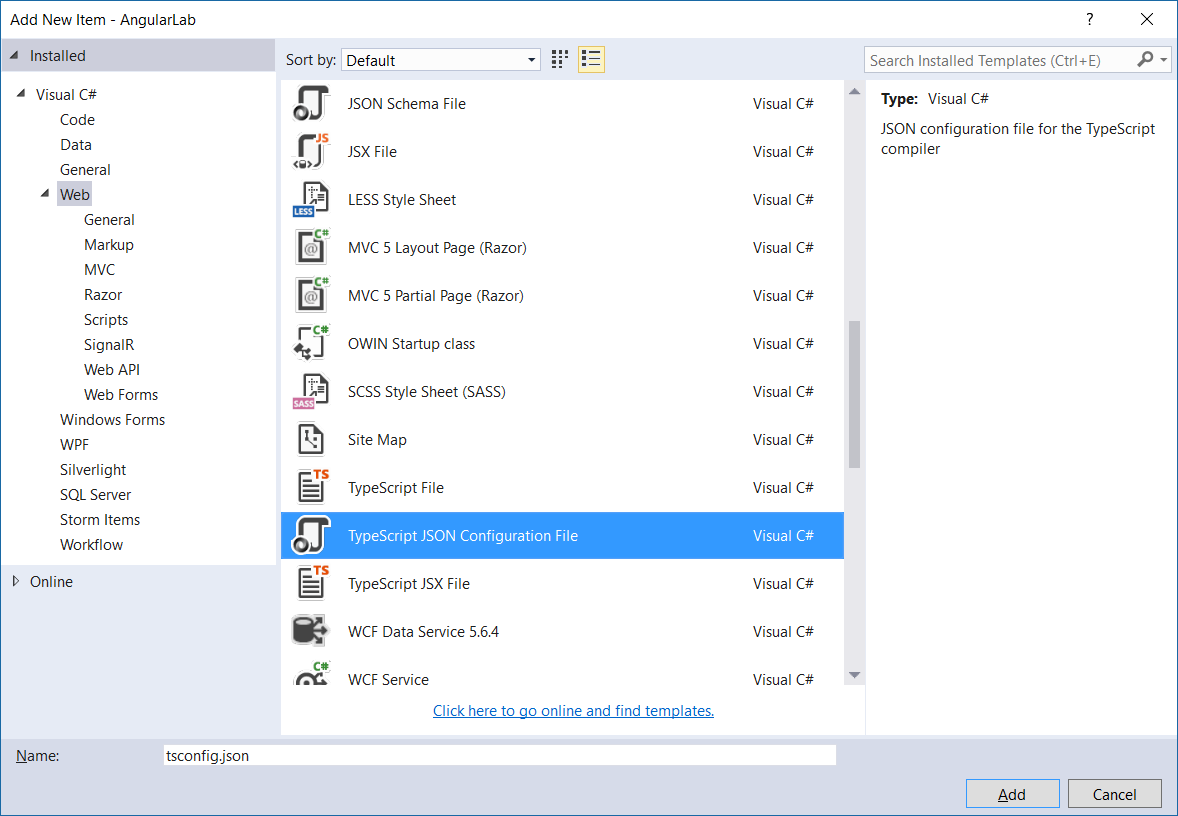
* Create a new simple SPA (Single Page Application) using ASP.NET MVC, HTML5 and the Angular 2 framework, demonstrating Angular’s MVC architecture.
* The app will be served up from an MVC web application, which will also implement a simple Web API call to return data from the server.

Task 1: Create the project, configure NPM packages and TypeScript

1. Create a new project in Visual Studio 2017, select ASP.NET MVC Web Application.
2. In Visual Studio 2017, create a new **ASP.NET Web Application**. Enter the name *AngularLab*, and click **OK**
3. In the **New ASP.NET Web Application** dialog box, select the **MVC** template, and keep the authentication settings to **No Authentication**, and then click **OK**.



1. In the **Solution Explorer** window, right-click to the project **AngularLab** and click **Add -> New Item**. Select **TypeScript JSON Configuration File** from the list and click **Add**.



1. Delete all the content in **tsconfig.json** file and copy the lines below instead. This configuration will allow you use decorators in Angular classes such as components, modules, services etc.

{

"compileOnSave": true,

"compilerOptions": {

"target": "es5",

"module": "commonjs",

"moduleResolution": "node",

"sourceMap": true,

"emitDecoratorMetadata": true,

"experimentalDecorators": true,

"removeComments": false,

"noImplicitAny": false

},

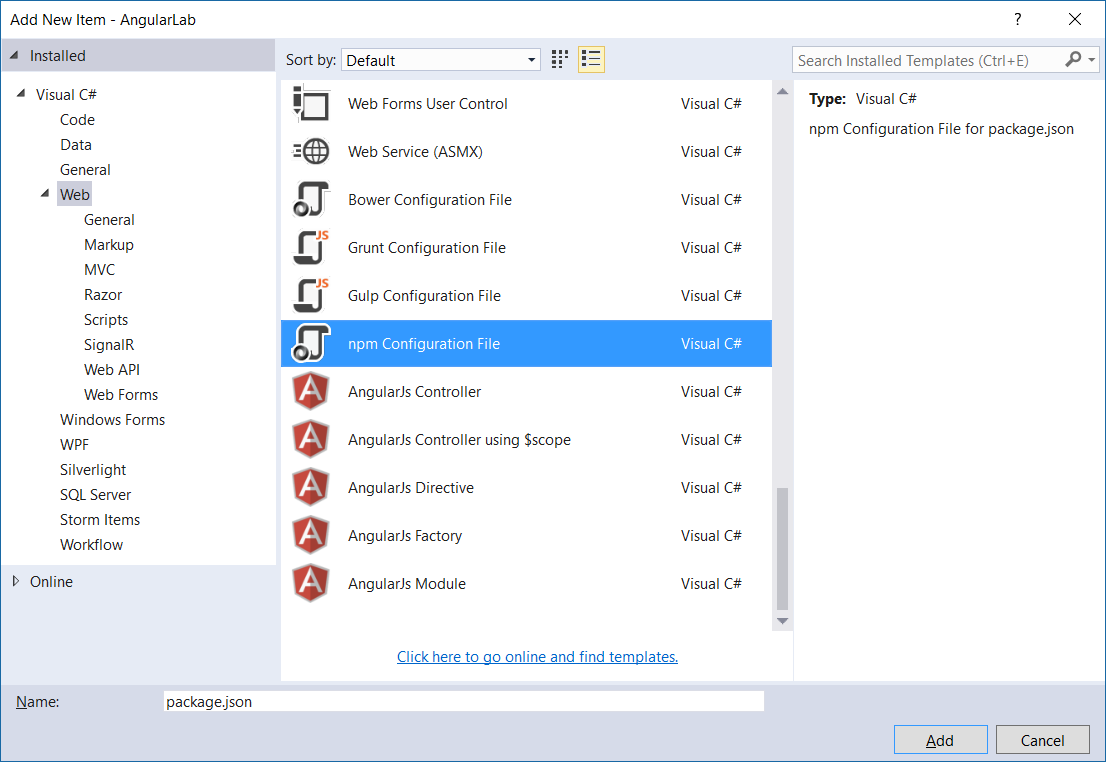
"exclude": [

"node\_modules"

]

}

1. Now there is one more step left to finish configuration of the project in order to successfully compile TypeScript files. In the **Solution Explorer** window, right-click to the project and click **Add -> New Item**. Select **npm Configuration File** from the list and click **Add**.



1. Open **package.json** file, and in **devDependencies** add type definitions for core-js standard library for JavaScript. In **dependencies** add Angular related packages. **package.json** file should look like the following:

{

"version": "1.0.0",

"name": "asp.net",

"private": true,

"dependencies": {

"@angular/common": "2.0.0",

"@angular/compiler": "2.0.0",

"@angular/core": "2.0.0",

"@angular/forms": "2.0.0",

"@angular/http": "2.0.0",

"@angular/platform-browser": "2.0.0",

"@angular/platform-browser-dynamic": "2.0.0",

"core-js": "^2.4.1",

"reflect-metadata": "^0.1.9",

"rxjs": "^5.0.3",

"systemjs": "^0.19.41",

"zone.js": "0.7.4"

},

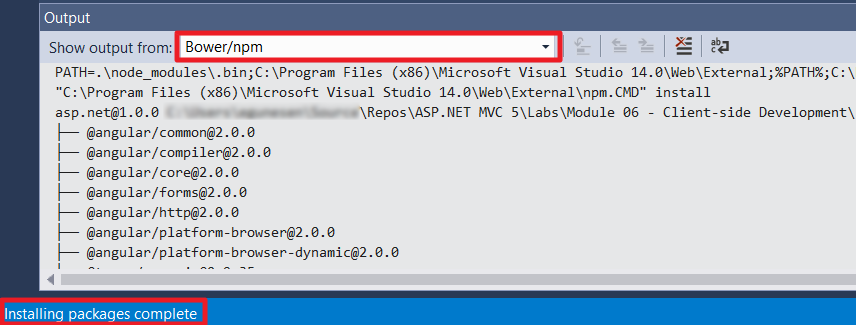
"devDependencies": {

"@types/core-js": "^0.9.35"

}

}

1. Save **package.json** file and wait for the packages installed. You can see the installation logs from **Bower/npm** section in **Output** window.



Task 2: Add the Web API Call to the Home Controller

1. Right-click the **Models** folder and click **Add -> Class**, and then enter **VehicleViewModel** for class name. This class will describe a vehicle, which for simplicity sake matches the model that is to be used in the client-side of the application. The class should look like the following:

public class VehicleViewModel

{

public string make { get; set; }

public string model { get; set; }

public string registration { get; set; }

public string colour { get; set; }

public string year { get; set; }

}

1. Open HomeController.cs file.
2. Add a new method somewhere in the class using the following code:

[HttpGet]

public ActionResult LoadVehicles()

{

return Json(new VehicleViewModel[]

{

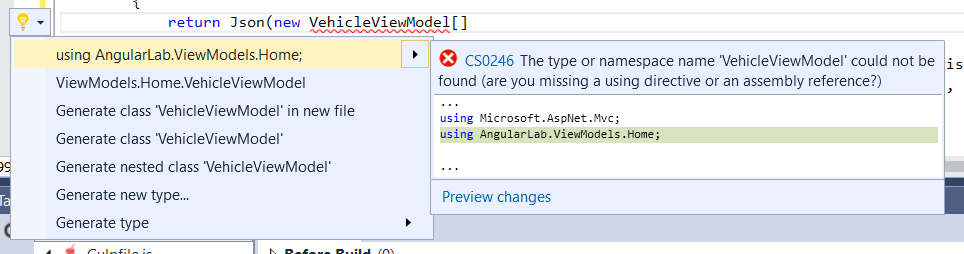
new VehicleViewModel() { make = "Ferrari", model = "California", colour="Lusso", registration="F45T 0NE", year="2012" },

new VehicleViewModel() { make = "TVR", model="Chimaera 500", colour ="Starmist Blue", registration = "R86 0EL", year="1998" }

}, JsonRequestBehavior.AllowGet);

}

1. Of course in a production application, this would probably use a database connection instead to load the relevant data – this is a very simplified example! You will need to resolve the missing reference by pressing **Ctrl** + **Space** on the word **VehicleViewModel**, and adding the recommended **using** statement.



Task 3: Setup Angular 2

1. In order to configure loading modules, you will add SystemJS configuration file. Right-click the project and click **Add -> JavaScript** file, and enter **system.config.js** as file’s name.
2. Copy the following lines in **system.config.js** file:

(function (global) {

System.config({

paths: {

// paths serve as alias

'npm:': 'node\_modules/'

},

// map tells the System loader where to look for things

map: {

// our app is within the app folder

app: 'app',

// angular bundles

'@angular/core': 'npm:@angular/core/bundles/core.umd.js',

'@angular/common': 'npm:@angular/common/bundles/common.umd.js',

'@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',

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

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

'@angular/http': 'npm:@angular/http/bundles/http.umd.js',

// other libraries

'rxjs': 'npm:rxjs'

},

// packages tells the System loader how to load when no filename and/or no extension

packages: {

app: {

main: './main.js',

defaultExtension: 'js'

},

rxjs: {

defaultExtension: 'js'

}

}

});

})(this);

**Note**: When organizing an application components into small individual files, it is better to use a module loader instead of loading all files with <script> tags. SystemJS is one of the module loaders which enables usage of built-in module systems of the TypeScript or ES2015 languages.

1. Now you need to add a reference to SystemJS configuration file and code to load the actual application in **app** folder (which you will add in next steps). Open **\_Layout.cshtml**, and then add the following lines just before the **</body>** closing tag:

<script src="/node\_modules/core-js/client/shim.min.js"></script>

<script src="/node\_modules/zone.js/dist/zone.js"></script>

<script src="/node\_modules/reflect-metadata/Reflect.js"></script>

<script src="/node\_modules/systemjs/dist/system.src.js"></script>

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

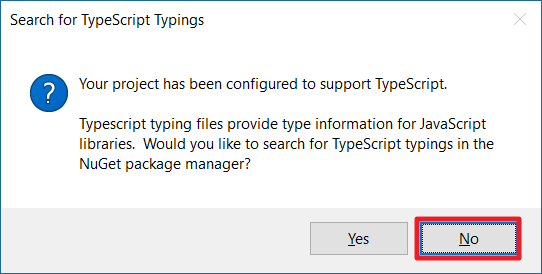
<script>

System.import('/app')

.catch(console.error.bind(console));

</script>

1. Right-click to the project, and then click **Add -> New Folder**. Enter **app** as folder name.
2. Right-click to app folder, and then click **Add -> TypeScript File**. Enter **main.ts** as file name. Click **No** if you are prompted for TypeScript Typings.



1. In **main.ts** file add the following codes to load AppModule which will be the root module of your application.

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);

Task 4: Time to add the Angular Client-side Application

1. Now all necessary steps are completed to bootstrap the application, you are ready to add application specific codes. You will need to add three assets:
2. Main angular app module.
3. App component which will display vehicles.
4. An angular service, that will be used to communicate with the web API.
5. Right-click to **app** folder, and then click **Add -> TypeScript File**. Enter **app.module.ts** as file name.
6. Add the code below in **app.module.ts** which will be the root module of your application.

import { NgModule } from '@angular/core';

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

import { AppComponent } from './app.component';

@NgModule({

imports: [BrowserModule],

declarations: [AppComponent],

bootstrap: [AppComponent]

})

export class AppModule { }

Notice that AppModule class is decorated with **@NgModule** attribute which enables definition and configuration of Angular 2 modules.

1. In the previous step, the root module has referenced a component named AppComponent. Now, to define AppComponent right-click to **app** folder, and then click **Add -> TypeScript File**. Enter **app.component.ts** as file name.
2. Add the following code in **app.component.ts** file:

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

import { Vehicle } from './vehicle';

@Component({

selector: 'vehicles',

templateUrl: 'app/app.component.html'

})

export class AppComponent {

pageTitle: string = `Vehicle Catalogue`;

errorMessage: string;

vehicles: Vehicle[];

}

1. AppComponent has **vehicles** property which is an array of **Vehicle** class. Now you need to add a TypeScript class to define vehicles. Right-click to **app** folder, and then select **Add** **> TypeScript File.** Enter **vehicle.ts** as file name.
2. Add the following code in **vehicle.ts** file:

export class Vehicle {

make: string;

model: string;

colour: string;

registration: string;

year: string;

}

1. Now AppComponent has all its properties ready, you can add its HTML template and define how it will be rendered. Right-click the **app** folder, and then click **Add -> HTML Page.** Enter **app.component.html** as file name.
2. Add the code below in **app.component.html** file. This template makes use of **\*ngFor** directive to create a table row for each vehicle. The other directive **\*ngIf** will enable rendering of div element if the **errorMessage** property of component is not *undefined.* Also with Interpolation, component properties (such as **pageTitle**) are rendered.

<h2>**{{**pageTitle**}}**</h2>

<table class="table table-striped">

<thead>

<tr>

<th>Registration</th>

<th>Make</th>

<th>Model</th>

<th>Colour</th>

<th>Year</th>

</tr>

</thead>

<tr \*ngFor='let vehicle of vehicles'>

<td>**{{**vehicle.registration**}}**</td>

<td>**{{**vehicle.make**}}**</td>

<td>**{{**vehicle.model**}}**</td>

<td>**{{**vehicle.colour**}}**</td>

<td>**{{**vehicle.year**}}**</td>

</tr>

</table>

<div \*ngIf="errorMessage">**{{**errorMessage**}}**</div>

1. Finally, since your Single Page Application will run on Home Page, open **Index.cshtml** file, delete all content, and then add the following lines:

@{

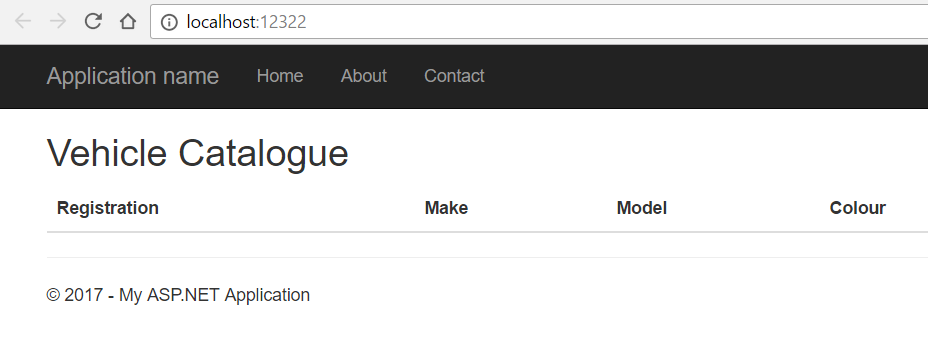
ViewBag.Title = "Home Page";

}

<vehicles>Loading vehicle catalogue...</vehicles>

Notice that **<vehicles>** tag is the same with **selector** attribute property of the AppComponent.

1. Now if you run the application, you will see an empty table since vehicle property is not filled yet. In the next task, you will get a vehicle list from HomeController.



Task 5: Inject Vehicle Service and make Web API Call

1. In order to get vehicle list from Web API call, you need to define a service which will be injected into the component. Right-click the **app** folder, and then click **Add -> TypeScript File.** Enter **vehicle.service.ts** as file name.
2. Add the following code in **vehicle.service.ts**:

import { Injectable } from '@angular/core';

import { Http, Response } from '@angular/http';

import { Observable } from 'rxjs/Observable';

import { Vehicle } from './vehicle';

import 'rxjs/add/operator/catch';

import 'rxjs/add/operator/map';

import 'rxjs/add/operator/do';

import 'rxjs/add/observable/throw';

@Injectable()

export class VehicleService {

private \_url = '/home/loadvehicles';

constructor(private \_http: Http) { }

getVehicles() {

return this.\_http.get(this.\_url)

.map((response: Response) => <Vehicle[]>response.json())

.do(data => console.log(data))

.catch(this.handleError);

}

private extractData(res: Response) {

return <Vehicle[]> res.json();

}

private handleError(error: Response) {

console.error(error);

let msg = `Error status code ${error.status} at ${error.url}`;

return Observable.throw(msg);

}

}

**Note**: Notice that Angular http.get method returns an RxJS Observable instead of a value type, such as Vehicle array. Observables are a powerful way to manage asynchronous data flows. For more information on Reactive Extensions, see: <http://reactivex.io/rxjs/manual/overview.html>

1. Now **VehicleService** is ready, it is time to use it in the AppComponent. Open **app.component.ts** file, import VehicleService, add it to providers list, and then add code to consume service methods. Finally, **app.component.ts** file should look like to:

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

import { Vehicle } from './vehicle';

import { VehicleService } from './vehicle.service';

@Component({

selector: 'vehicles',

templateUrl: 'app/app.component.html',

providers: [VehicleService]

})

export class AppComponent {

pageTitle: string = `Vehicle Catalogue`;

errorMessage: string;

vehicles: Vehicle[];

constructor(private vehicleService: VehicleService) { }

ngOnInit() {

this.vehicleService.getVehicles()

.subscribe(

vehicles => this.vehicles = vehicles,

error => this.errorMessage = <any>error

);

};

}

Notice that there is no code which instantiates VehicleService class. It is injected through the constructor when the component is created. Notice also that getVehicles method of the service returns an Observable object the component is subscribed to.

1. Since the VehicleService class will be injected to the component, the root module should be aware of the VehicleService provider. Open **app.module.ts** file, import HttpModule and VehicleService. Finally, **app.module.ts** file should look like to:

import { NgModule } from '@angular/core';

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

import { AppComponent } from './app.component';

import { HttpModule } from '@angular/http';

import { VehicleService } from './vehicle.service';

@NgModule({

imports: [BrowserModule, HttpModule],

declarations: [AppComponent],

providers: [VehicleService],

bootstrap: [AppComponent]

})

export class AppModule { }

1. Run the application again and see that the vehicle list is loaded from the Web API method you have defined in HomeController.cs.

