

Developing a Web App using TypeScript and AngularJS 1.5

Lab Time: 45 minutes

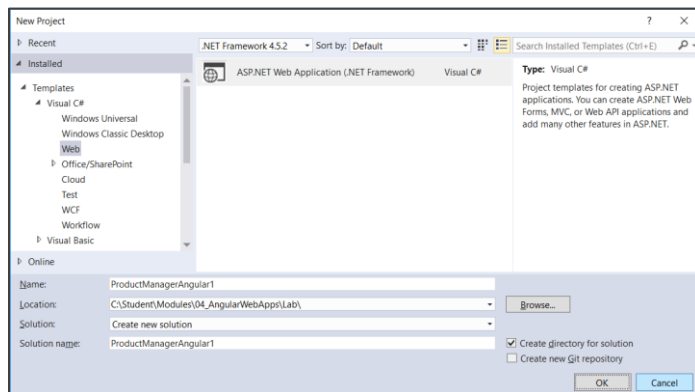
Lab Folder: C:\Student\Modules\04_AngularWebApps\Lab

Lab Overview: In this lab you will get hands-on experience working with version 1.5 of the AngularJS framework by developing a web app which is based on client-side code written using TypeScript. You will build the project for this Angular web app from the ground up using Visual Studio 2017. Working through these set of lab exercises will help to build your understanding of how routing, controllers and view templates fit into the big picture of developing with the AngularJS framework.

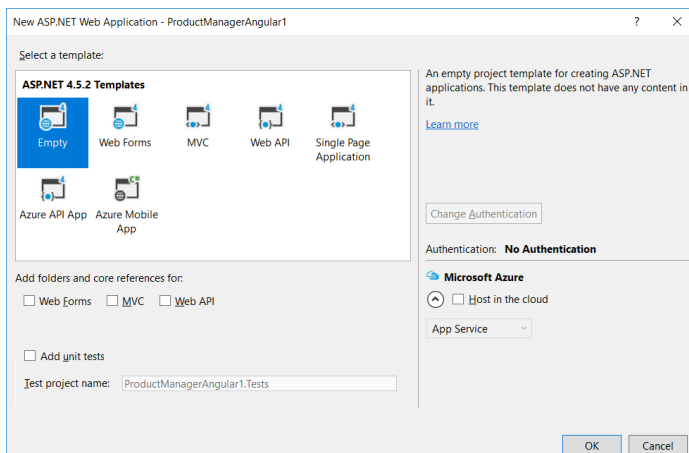
Exercise 1: Create a Visual Studio Project using Bootstrap and AngularJS

In this exercise you will create a new ASP.NET Web Application project in Visual studio and configure it to use the bootstrap CSS framework and the AngularJS framework.

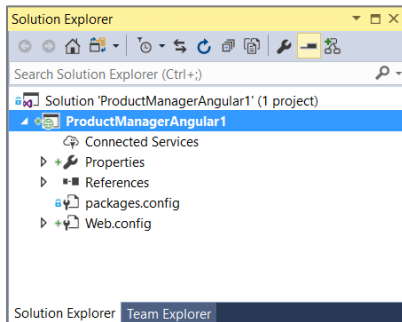
1. Launch **Visual Studio 2017**.
2. Create the new solution in Visual Studio 2017:
 - a) Create a new project by selecting the menu command **File > New > Project**.
 - b) In the **New Project** dialog...
 - i) Select the **ASP.NET Web Application** project template under the **Templates > Visual C# > Web** section.
 - ii) Enter a name of **ProductManagerAngular1**.
 - iii) Enter a location of **C:\Student\Modules\04_AngularWebApps\Lab**.
 - iv) Click the **OK** button.



- c) In the New **ASP.NET Web Application** dialog, select the **Empty** project template.
- d) Make sure **Authentication** is set to **No Authentication**.
- e) Make sure the **Host in the cloud** checkbox is unchecked.
- f) Click the **OK** button to create the new project.



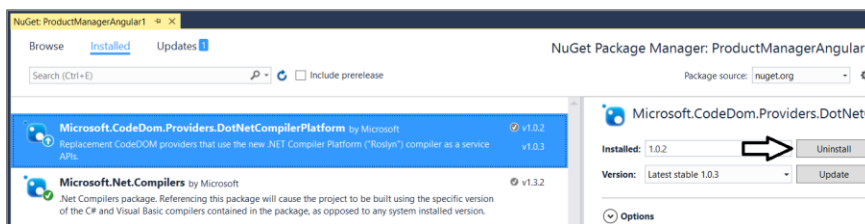
3. Once Visual Studio has created the project, familiarize yourself with the structure of the project in the **Solution Explorer**. As you can see the project doesn't contain many files at this point.



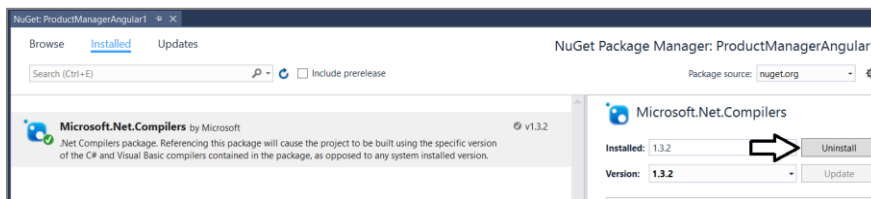
4. Configure the **ProductManagerAngular1** project with the required set of NuGet packages.
- a) Right-click on the **ProductManagerAngular1** project in the Solution Explorer and select **Manage NuGet Packages**.

Using the **Manage NuGet Packages** dialog, you should be able to see that two packages were added to your project when it was created. Over the next steps, you will uninstall these two packages.

- b) Uninstall the **Microsoft.CodeDom.Providers.DotNetCompilerPlatform** package.



- c) Uninstall the **Microsoft.Net.Compilers** package.



Technically, you do not have to uninstall these two packages. However, the purpose of this lab is to write a web app using only client-side code. There will be no need to compile any .NET code on the server so these NuGet packages are not required.

- d) Install the latest version of the **bootstrap** package.

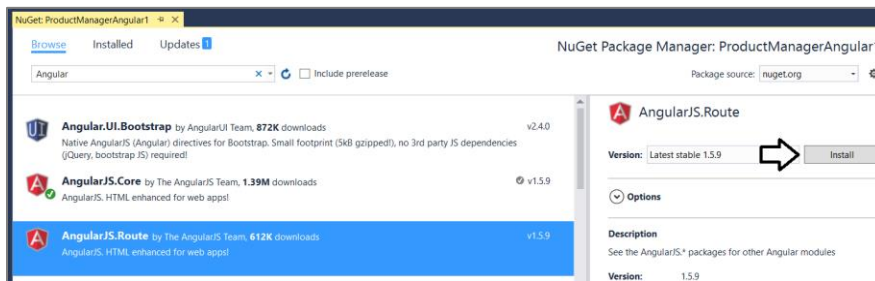


When you install the **bootstrap** package, the NuGet Packager Manager will automatically install the **jQuery** package because the **bootstrap** package has been defined with a dependency on the **jQuery** package.

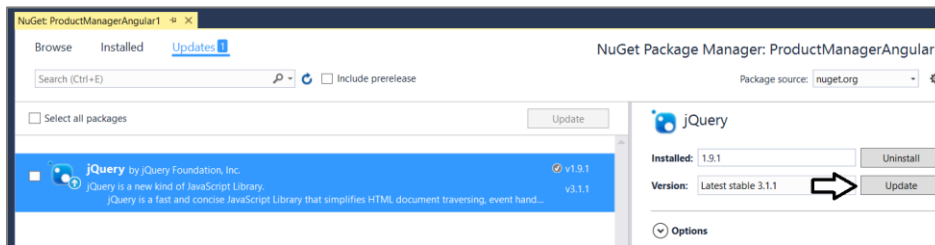
- e) Install the latest version of the **AngularJS.Core** package that is greater than or equal to version 1.5 but less than version 2.0.



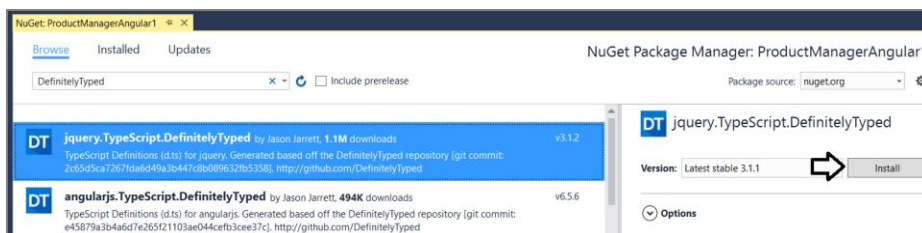
- f) Install the **AngularJS.Route** package using the same version you used for the **AngularJS.Core** package.



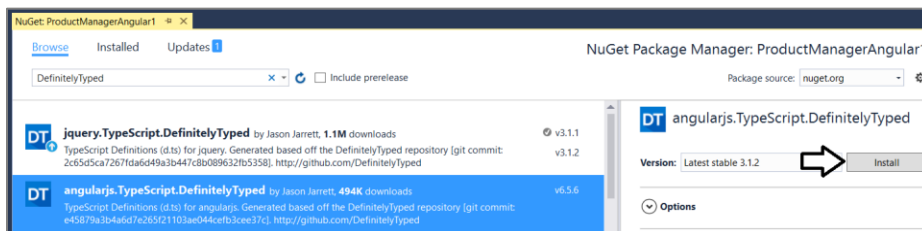
- g) Move to the **Updates** tab and update the **jQuery** package to the latest version.



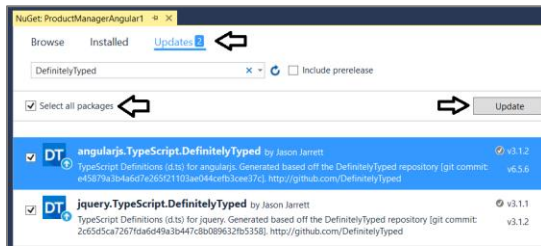
- h) Install the latest version of the **jquery.TypeScript.DefinitelyTyped** package.



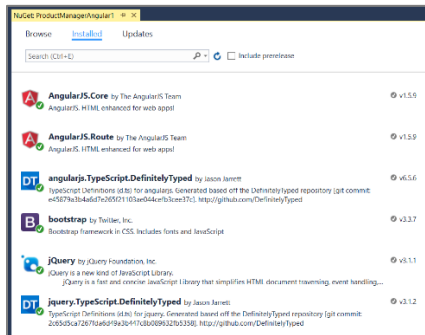
- i) Install the latest version of the **angularjs.TypeScript.DefinitelyTyped** package.



- j) Inspect the **Updates** tab. If any updates are available, update both **DefinitelyTyped** packages to the latest version.

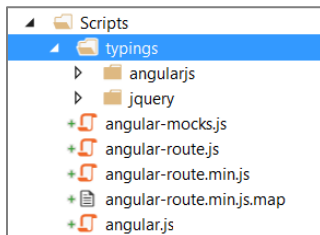


- k) The set of packages in your project should now match the set of packages shown in the following screenshot.

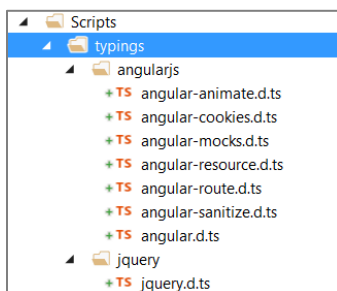


You have now installed all the NuGet packages and 3rd party libraries you will need to develop the **ProductManagerAngular1** project. Note that while you have used the NuGet package manager, you could have installed the same set of libraries with the new support in Visual Studio for adding Bower packages as well.

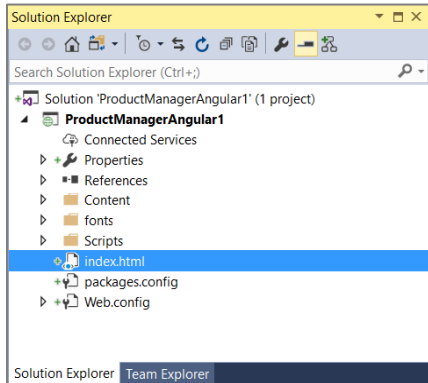
5. Examine the 3rd part package files that have been added to the project.
- a) In Solution Explorer, expand the **Content** folder and the **fonts** folder to see the files added from the **bootstrap** package.
 - b) Expand the **Scripts** folder to see the files for the JavaScript library packages.
 - c) Expand the **Scripts/typings** folder. You should see folders for the DefinitelyTyped packages created for jQuery and Angular.



- d) If you expand these two folders, you will see the **d.ts** files containing the typed definitions for Angular and jQuery.



6. Add a new HTML file named **index.html** to the root folder.
 - a) In Solution Explorer, right-click the top-level project node and select **Add > HTML Page**.
 - b) Enter an **Item name** of **index.html**.
 - c) You should now see **index.html** in the Solution Explorer in the root folder of the **ProductManagerAngular1** project.



- d) Copy and paste the following starter HTML code into **index.html**.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Product Manager NG</title>
  <link href="/Content/bootstrap.css" rel="stylesheet" />
  <!-- required by Angular routing -->
  <base href="/" />
</head>
<body>

  <!-- add top navbar here -->

  <!-- add main body content here -->

  <!-- add script links to 3rd party libraries -->

  <!-- add script links to internal JS files -->

</body>
</html>
```

- e) Locate the **add top navbar here** comment in **index.html** and then paste the following HTML code below it.

```
<!-- add top here navbar -->
<div class="container">
  <div class="navbar navbar-default " role="navigation">
    <div class="container-fluid">
      <div class="navbar-header">
        <a class="navbar-brand" href="/">Product Manager Angular</a>
      </div>
      <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
          <li><a href="/products">Products List</a></li>
          <li><a href="/products/showcase">Product Showcase</a></li>
        </ul>
      </div>
    </div>
  </div>
</div>
```

- f) Locate the **add main body content here** comment in **index.html** and then paste the following HTML code below it.

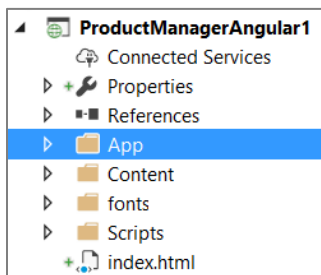
```
<!-- add main body content here -->
<div class="container">
  <div class="container-fluid">
    <div id="content-box"></div>
  </div>
</div>
```

- g) Locate the **add script links to 3rd party libraries** comment and add links to jQuery, bootstrap, angular and angular-route.

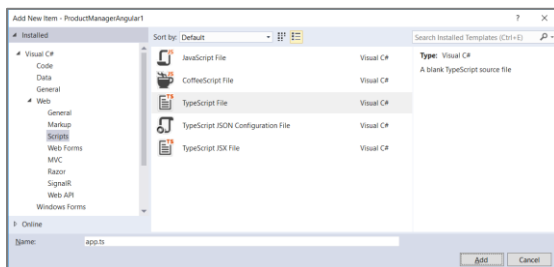
```
<!-- add script links to 3rd party libraries -->
<script src="Scripts/jquery-3.1.1.js"></script>
<script src="Scripts/bootstrap.js"></script>
<script src="Scripts/angular.js"></script>
<script src="Scripts/angular-route.js"></script>
```

Make sure the version number in the link to the jQuery library matches the version number of the jQuery package in your project.

7. Create a new top-level folder in the **ProductManagerAngular1** project named **App**.



8. Create a new TypeScript file in the **App** folder named **app.ts**.
- Right-click on **App** folder and select the **Add > New Item** menu command.
 - On the left side of the **New Item** dialog, select **Visual C# > Web > Scripts**.
 - Select **TypeScript File** as the item template and then click the **Add** button in the bottom right of the dialog.



9. Implement the traditional "hello world" application code in **app.ts**.
- Add the following code in **app.ts** to create a startup function using the jQuery document ready event.

```
module myApp {
  $( function() {
    // initialize the app
  });
}
```

Wait just a second. Don't use the same old boring JavaScript syntax to create an anonymous function. TypeScript developers will consider you "old school". Instead, use the trendy new arrow function syntax to show folks how much you love TypeScript.

- b) Modify your code to use TypeScript arrow function syntax as shown in the following code listing.

```
module myApp {  
    $( () => {  
        // initialize the app  
    });  
}
```

- c) Create a variable named **contentBoxDiv** and initialize it with a jQuery object which wraps the div element in **index.html** which has an id of **content-box**. Make sure to declare the **contentBoxDiv** variable so it is strongly-typed to the **jQuery** type.

```
// initialize the app  
let contentBoxDiv: JQuery = $("#content-box");
```

- d) Execute the **text()** method and the **css()** method on the **contentBoxDiv** object as shown in the following code listing.

```
module myApp {  
    $( () => {  
        // initialize the app  
        let contentBoxDiv: JQuery = $("#content-box");  
  
        contentBoxDiv  
            .text("Hello TypeScript")  
            .css({  
                "color": "Blue",  
                "font-size": "32px"  
            });  
    });  
}
```

- e) Save your changes to **app.ts**.

10. Update **index.html** with a script link to **app.js**.

- a) Open **index.html** in an editor window if it is not already open.
b) Add the following script link to **app.js** at the bottom of the page.

```
<!-- Add script links to internal JS files -->  
<script src="App/app.js"></script>
```

- c) The code at the bottom of **index.html** should match the following screenshot.

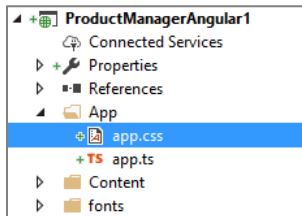
```
<!-- Add script links to 3rd party libraries -->  
<script src="Scripts/jquery-3.1.1.js"></script>  
<script src="Scripts/bootstrap.js"></script>  
<script src="Scripts/angular.js"></script>  
<script src="Scripts/angular-route.js"></script>  
  
<!-- Add script links to internal JS files -->  
<script src="App/app.js"></script>  
  
</body>  
</html>
```

11. Import the CSS file named **app.css** into your project to provide the CSS styles you will use in this lab.

- a) Right-click on the **App** folder and select the **Add > Existing Item...** command.
b) When prompted for a file path, enter the following path to the CSS source file named **app.css**.

```
C:\Student\Modules\04_AngularWebApps\Lab\StarterFiles\app.css
```

- c) The source file named **app.css** should now be located inside the **App** folder alongside **app.ts**.



- d) Update **index.html** with stylesheet link to the **app.css** file.

```
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Product Manager NG</title>
  <link href="/Content/bootstrap.css" rel="stylesheet" />
  <link href="/App/app.css" rel="stylesheet" />
  <!-- required by Angular routing -->
  <base href="/" />
</head>
```

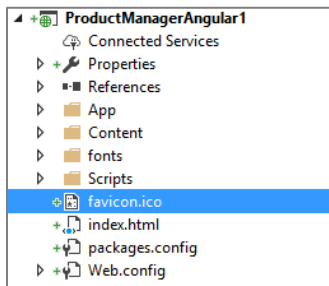
- e) Save your changes to **index.html**.

12. Add a **favicon.ico** file to the root folder of the **ProductManagerMVC** project.

- a) Using Windows Explorer, locate the file named **favicon.ico** in the **Students** folder at the following location.

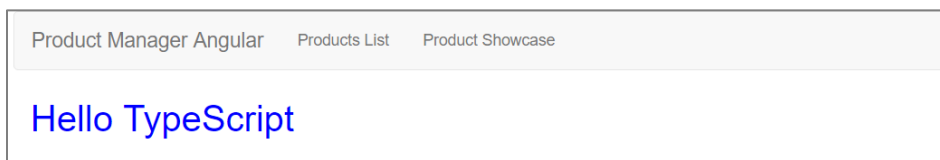
C:\Student\Modules\04_AngularWebApps\Lab\StarterFiles\favicon.ico

- b) Copy the file named **favicon.ico** to the root folder of your project.



13. Test out the **ProductManagerAngular1** project using the Visual Studio Debugger

- a) Press the **{F5}** key to start up the project in the Visual Studio debugger.
b) When the project starts, it should appear in the browser and match the following screenshot.



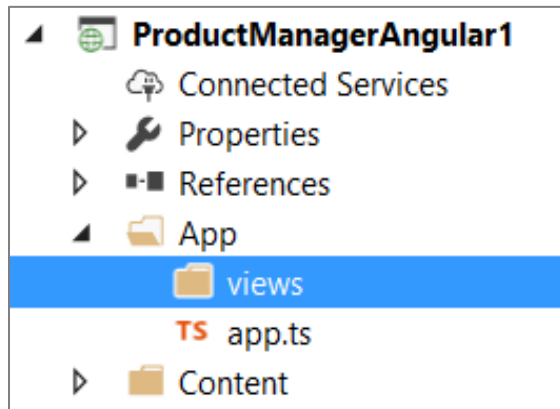
- c) Once you have tested **ProductManagerAngular1**, close the browser window, return to Visual Studio and stop the debugger.

In this initial exercise, you got your project up and running and you have begun to write and test code written in TypeScript. You have also seen that typed definitions make it possible to program in TypeScript against JavaScript libraries such as jQuery in a strongly typed fashion. From this point on in this lab, you will not program against the jQuery library anymore. Instead, you will now only program against the typed definitions for **angular.js** and **angular-route.js**. But now when you need to program against the jQuery library in other projects where you are using TypeScript, you know how to get started.

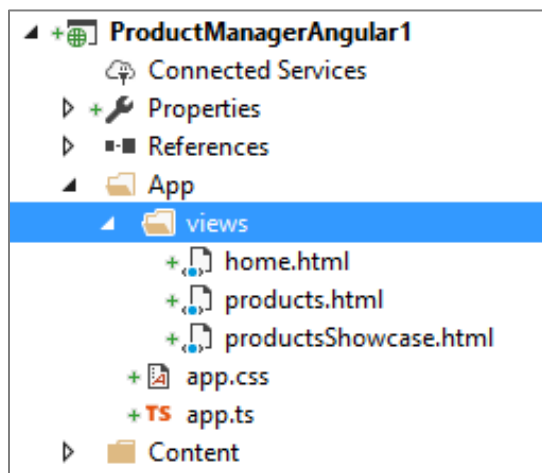
Exercise 2: Configure the Project with Angular Routes, Views and Controllers

In this exercise you will write code to initialize the **ProductManagerAngular1** web app as an Angular application by creating a routing involving view templates and controllers. Along the way, you will modify the **Web.config** file to add the server-side URL rewriting support which is required when using the Angular framework in HTML5 mode which eliminates the need to add a # into routing URLs.

1. Create the initial set of view templates.
 - a) Create a new folder inside the **App** folder with the name of **views**.



- b) Add three new HTML files to the **views** folder named **home.html**, **products.html** and **productShowcase.html**.



- c) Delete any content inside **home.html** and replace it with the following HTML code.

```
<h2>Home</h2>
```

- d) Delete any content inside **products.html** and replace it with the following HTML code.

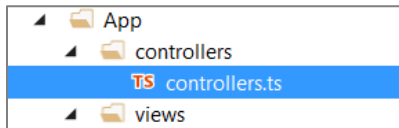
```
<h2>Products List</h2>
```

- e) Delete any content inside **productsShowcase.html** and replace it with the following HTML code.

```
<h2>Product Showcase</h2>
```

- f) Save your changes and close all three view template files.
2. Create a new TypeScript source file for the project's controllers.
 - a) Create a new folder inside the **App** folder with the name of **controllers**.

- b) Add a new TypeScript file inside the **controllers** folder named **controllers.ts**.



- c) Add the following code to **controllers.js** to provide a starting point for three generic Angular controller classes.

```
module myApp {  
    let app = angular.module("myApp");  
  
    class HomeController {  
        static $inject: Array<string> = [];  
        constructor() { }  
    }  
  
    app.controller('homeController', HomeController);  
  
    class ProductsController {  
        static $inject: Array<string> = [];  
        constructor() {}  
    }  
  
    app.controller('productsController', ProductsController);  
  
    class ProductShowcaseController {  
        static $inject: Array<string> = [];  
        constructor() { }  
    }  
  
    app.controller('productShowcaseController', ProductShowcaseController);  
}
```

- d) Save your changes to **controllers.ts**.

Your next step is to modify **index.html** with a few Angular directives.

3. Update **index.html** with the **ng-app** directive and the **ng-view** directive.

- a) Open **index.html** in a code editor window if it is not already open.
b) Locate the opening tag for the HTML **body** element and add the **ng-app** directive with a value of **myApp**.

```
<body ng-app="myApp">
```

- c) Locate the opening tag for the div element with the **id** of **content-box** and add the **ng-view** directive.

```
<div class="container">  
    <div class="container-fluid">  
        <div id="content-box" ng-view ></div>  
    </div>  
</div>
```

4. Update **index.html** with a script link to **controllers.js**.

- a) Move to the bottom of **index.html** and add a script link to **controllers.js** underneath the existing script link to **app.js**.

```
<!-- Add script links to internal JS files -->  
<script src="App/app.js"></script>  
<script src="App/controllers/controllers.js"></script>
```

- b) Save your changes to **index.html**.

5. Update the code in **app.ts** to initialize the app with a set of Angular routes.

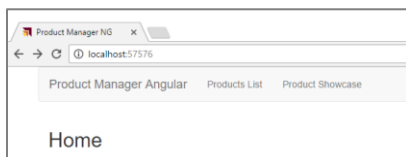
- a) Open **app.ts** in a code editor window if it is not already open.

- b) Remove all the existing code inside **app.ts** and replace it with the following code.

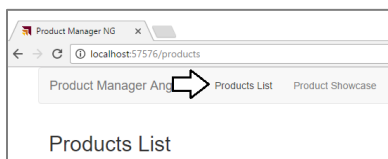
```
module myApp {  
  
    var app = angular.module("myApp", ['ngRoute']);  
  
    app.config( ($locationProvider: ng.ILocationProvider,  
                $routeProvider: ng.route.IRouteProvider) => {  
  
        $locationProvider.html5Mode(true);  
  
        $routeProvider  
            .when("/", {  
                templateUrl: 'App/views/home.html',  
                controller: "homeController",  
                controllerAs: "vm"  
            })  
            .when("/products", {  
                templateUrl: 'App/views/products.html',  
                controller: "productsController",  
                controllerAs: "vm"  
            })  
            .when("/products/showcase", {  
                templateUrl: 'App/views/productsShowcase.html',  
                controller: "productShowcaseController",  
                controllerAs: "vm"  
            })  
            .otherwise({ redirectTo: "/" });  
    });  
}
```

The call to **\$locationProvider.html5Mode(true)** configures the application to run in HTML5 mode eliminating the need to add a "#" character into each routing URL. Calls to **\$routeProvider.when()** set up routes in the application's routing scheme. Also, the **ControllerAs** attribute on a route makes it possible to access the view model from inside view templates using **vm** eliminating the need to pass the view model to the view template using the **\$scope** service which was popular in the early days of Angular.

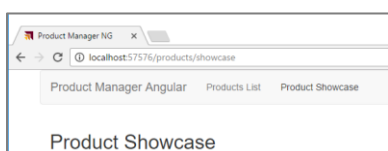
6. Test out the **ProductManagerAngular1** project using the Visual Studio Debugger
- Press the **{F5}** key to start up the project in the Visual Studio debugger.
 - When the project starts, you should see the home page with its route at the application root.



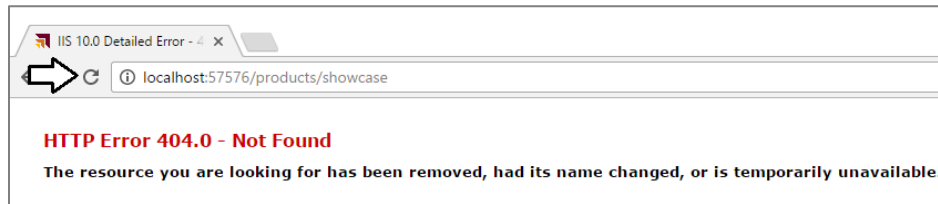
- c) Click on the **Products List** link to navigate to the **/products** route.



- d) Click on the **Product Showcase** link to navigate to the **/products/showcase** route.

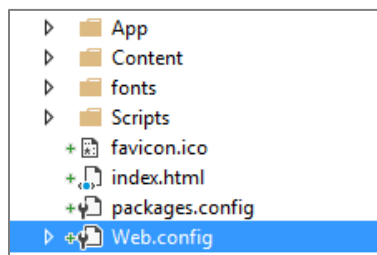


- e) While located at the **/products/showcase** route, click the **Refresh** button in the browser and you will experience a 404 error.



This 404 error reveals an important aspect of working in HTML5 mode. You need to add server-side support for URL rewriting.

7. Add server-side support for URL rewriting into the application's **Web.config** file.
- a) In Solution Explorer, double-click on the **Web.config** file to open it in an editor window.



- b) Replace the contents of **Web.config** with the following XML code.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>

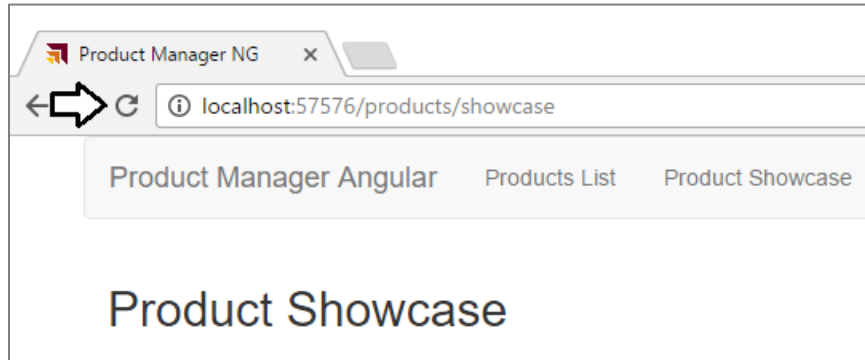
  <system.web>
    <compilation debug="true" targetFramework="4.5.2"/>
    <httpRuntime targetFramework="4.5.2"/>
  </system.web>

  <system.webServer>
    <rewrite>
      <rules>
        <rule name="AngularJS Routes" stopProcessing="true">
          <match url=".*" />
          <conditions logicalGrouping="MatchAll">
            <add input="{REQUEST_FILENAME}" pattern="(.*?)\.html$" negate="true" />
            <add input="{REQUEST_FILENAME}" pattern="(.*?)\.js$" negate="true" />
            <add input="{REQUEST_FILENAME}" pattern="(.*?)\.css$" negate="true" />
            <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
            <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
            <add input="{REQUEST_URI}" pattern="^(/api)" negate="true" />
          </conditions>
          <action type="Rewrite" url="/" />
        </rule>
      </rules>
    </rewrite>
  </system.webServer>
</configuration>
```

- c) Save your changes and close the **Web.config** file.

Web.config has now been updated with server-side URL rewriting support for a typical Angular application running in HTML5 mode. When the web server receives an incoming request for an Angular routing URL such as **/products** or **/products/showcase**, the request is rerouted to the root of the application and the web server returns the **index.html** page back to the browser. However, when the web server receives an incoming request to a URL for one of the application's source files such as **app.js**, **app.css** or **products.html**, the web server returns the source file back to the browser in a standard fashion.

8. Test out the **ProductManagerAngular1** project using the Visual Studio Debugger
 - a) Press the **{F5}** key to start up the project in the Visual Studio debugger.
 - b) Click on the **Product Showcase** link to navigate to the **/products/showcase** route.
 - c) Click the **Refresh** button in the browser and verify that you no longer experience a 404 error due to server-side URL rewriting.



9. Modify the **HomeController** class to be a view model which passes state for the home page to the underlying view template.
 - a) Open **controllers.ts** and replace the existing implementation of the **HomeController** class with the following code.

```
class HomeController {
    static $inject: Array<string> = [];
    welcomeMessage = "Welcome to the Wingtip Product Manager";
    topic1Title = "Add a new product";
    topic1Copy = "Click the Add Product link on the navbar above to add a new product.";
    topic2Title = "See the Product Showcase";
    topic2Copy = "Click the Product Showcase link above to see the full set of wingtip products.";
    constructor() { }
}
```

- b) Save your changes to **controllers.ts**.
10. Modify the view template named **home.html** to declaratively bind to data in the underlying view model.
 - a) Double click on the **home.html** view template file in the **App > views** folder to open it in an editor window.
 - b) Replace the contents of **home.html** with the following HTML code.

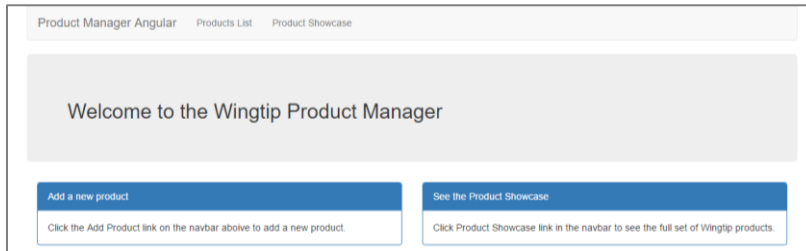
```
<div class="row">
  <div class="jumbotron">
    <h2 ng-bind="vm.welcomeMessage"></h2>
  </div>
</div>

<div class="row">
  <div class="col-md-6">
    <div class="panel panel-primary">
      <div ng-bind="vm.topic1Title" class="panel-heading"></div>
      <div ng-bind="vm.topic1Copy" class="panel-body"></div>
    </div>
  </div>
  <div class="col-md-6">
    <div class="panel panel-primary">
      <div ng-bind="vm.topic2Title" class="panel-heading"></div>
      <div ng-bind="vm.topic2Copy" class="panel-body"></div>
    </div>
  </div>
</div>
```

- c) Save your changes and close **home.html**.

You can see that this code references the view model in the **ng-bind** attribute setting using the **vm** view model prefix.

11. Test out the **ProductManagerAngular1** project using the Visual Studio Debugger
 - a) Press the **{F5}** key to start up the project in the Visual Studio debugger.
 - b) The home page should now display new content and match the page shown in the following screenshot.

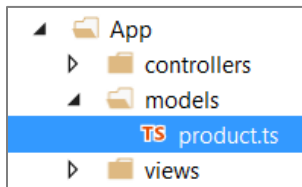


- c) Once you have tested **ProductManagerAngular1**, close the browser window, return to Visual Studio and stop the debugger.

Exercise 3: Use an Interface to Decouple Angular Controllers from Data Access Code

In this exercise you will add the data access code to manage a set of product data. You will begin by using TypeScript to create a class named **Product** and an interface named **IProductDataService** which will make it possible to write code in an Angular controller that is not tightly coupled to any data access code. In this exercise, you will also add a set of view templates to create HTML forms with Angular-style validation to provide a user interface experience with classic CRUD (*create-read-update-delete*) functionality.

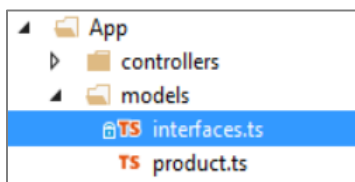
1. Create the **Product** class which will be used to design a strongly-typed view model containing product data.
 - a) In Solution Explorer, create a new folder named **models** inside the **App** folder.
 - b) Add a new TypeScript source file named **product.ts**.



- c) Implement the **Product** class by adding the following class definition to **Product.ts**.

```
module myApp {  
  export class Product {  
    Id: number;  
    Name: string;  
    Category: string;  
    ListPrice: number;  
    Description: string;  
    ProductImageUrl: string;  
  }  
}
```

- d) Save and close **product.ts**.
2. Add a new interface named **IProductDataService** to decouple controllers from the services that read and write product data.
 - a) Inside the **models** folder, add a new TypeScript source file named **interfaces.ts**.

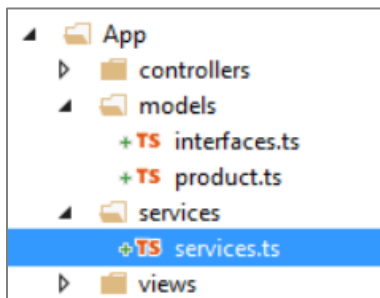


- b) Implement the **IProductDataService** class by adding the following code to **interfaces.ts**.

```
module myApp {  
  
  export interface IProductDataService {  
    GetAllProducts(): Product[];  
    GetProduct(id: number): Product;  
    AddProduct(product: Product): void;  
    DeleteProduct(id: number): void;  
    UpdateProduct(product: Product): void;  
    GetProductCategories(): string[];  
    GetProductsByCategory(category: string): Product[];  
  }  
  
}
```

Now that you have created the **IProductDataService** interface, you will create a class named **InMemoryProductDataService** which implements this interface. You will implement the **InMemoryProductDataService** class to store and manage a list of products in memory inside the current browser session. This type of design which relies on state behind a browser page isn't practical for most real-world applications because the data is never written into persistent storage. However, implementing this class and seeing how it fits into an Angular application as a whole will be a valuable learning experience. Also keep in mind that in later labs you will develop a similar Angular application that interacts with a web service to read and write product data to and from an Azure SQL Server database.

3. Create a new class named **InMemoryProductDataService** which will be used to create an Angular service.
- a) In Solution Explorer, create a new folder named **service** inside the **App** folder.
 - b) Add a new TypeScript source file named **services.ts**.



- c) Copy and paste the following code into **services.ts** to create the starting point for the **InMemoryProductDataService** class which will be designed to implement the **IProductDataService** interface.

```
module myApp {  
  
  export class InMemoryProductDataService implements IProductDataService {  
  
    static $inject: string[] = [];  
  
    GetAllProducts(): Product[] { /* not implemented yet */ };  
  
    GetProduct(id: number): Product { /* not implemented yet */ };  
  
    AddProduct(product: Product): void { /* not implemented yet */ };  
  
    DeleteProduct(id: number): void { /* not implemented yet */ };  
  
    UpdateProduct(product: Product): void { /* not implemented yet */ };  
  
    GetProductCategories(): string[] { /* not implemented yet */ };  
  
    GetProductsByCategory(category: string): Product[] { /* not implemented yet */ }  
  }  
  
}
```

In the next step you will copy-and-paste code from a pre-provided text file to create a list of product data. This will give you a set of products to test out your application without you having to do too much typing.

4. Create a field named **productListSeedData** which holds a set of sample product data for testing.
 - a) Using Windows Explorer, locate the file named **ProductListSeedData.cs.txt** in the **Students** folder at the following location.

```
C:\Student\Modules\04_AngularWebApps\Lab\StarterFiles\ProductListSeedData.ts.txt
```
 - b) Double-click on **ProductListSeedData.ts.txt** to open this file in Notepad.
 - c) Copy the contents of **ProductListSeedData.ts.txt** to the Windows clipboard.
 - d) Return to Visual Studio and place the cursor inside the **InMemoryProductDataService** class just underneath the **\$inject** field.
 - e) Paste the content of the Windows clipboard into the **InMemoryProductDataService** class at the cursor location.
 - f) Your class should now contain a new private field named **productListSeedData** as shown in the following screenshot.

```
export class InMemoryProductDataService implements IProductDataService {  
  
    static $inject: string[] = [];  
  
    private productListSeedData: Product[] = [...],  
    { Id: 2, Name: "Captain America Action Figure", ListPrice: 12.95, Category: "Action Figures", Description: "A super action f  
    { Id: 3, Name: "Easel with Supply Trays", ListPrice: 49.95, Category: "Arts and Crafts", Description: "A serious easel for s  
    { Id: 4, Name: "Crate o' Crayons", ListPrice: 14.95, Category: "Arts and Crafts", Description: "More crayons that you can sh  
    { Id: 5, Name: "Green Stomper Bully", ListPrice: 24.95, Category: "Remote Control", Description: "A green alternative to cru  
    { Id: 6, Name: "Indy Race Car", ListPrice: 19.95, Category: "Remote Control", Description: "The fastest remote control race  
    { Id: 7, Name: "Twitter Follower Action Figure", ListPrice: 1.00, Category: "Action Figures", Description: "An inexpensive a  
    { Id: 8, Name: "Sandpiper Prop Plane", ListPrice: 24.95, Category: "Remote Control", Description: "A simple RC prop plane fo  
    { Id: 9, Name: "Etch A Sketch", ListPrice: 12.95, Category: "Arts and Crafts", Description: "A strategic planning tool for t  
    { Id: 10, Name: "Flying Squirrel", ListPrice: 69.95, Category: "Remote Control", Description: "A stealthy remote control pla  
    { Id: 11, Name: "FOX News Chopper", ListPrice: 29.95, Category: "Remote Control", Description: "A new chopper which can gene  
    { Id: 12, Name: "Godzilla Action Figure", ListPrice: 19.95, Category: "Action Figures", Description: "The classic and adorab  
    { Id: 13, Name: "Perry the Platypus Action Figure", ListPrice: 21.95, Category: "Action Figures", Description: "A platypus w  
    { Id: 14, Name: "Seal Team 6 Helicopter", ListPrice: 59.95, Category: "Remote Control", Description: "A serious helicopter t  
    { Id: 15, Name: "Crayloa Crayon Set", ListPrice: 2.49, Category: "Arts and Crafts", Description: "A very fun set of crayons  
};  
  
    GetAllProducts(): Product[] { /* not implemented yet */ };
```

- g) Underneath the **productListSeedData** field, add a constructor to the class using the following code

```
constructor(private products: Product[]) {  
    this.products = this.productListSeedData;  
}
```

- h) At this point, the code at the top of **InMemoryProductDataService** should match the following screenshot.

```
module myApp {  
  
    export class InMemoryProductDataService implements IProductDataService {  
  
        static $inject: string[] = [];  
  
        private productListSeedData: Product[] = [...];  
  
        constructor(private products: Product[]) {  
            this.products = this.productListSeedData;  
        }  
  
        GetAllProducts(): Product[] { /* not implemented yet */ };
```

Your next set of tasks is to implement all the methods defined inside the **IProductDataService** interface.

- i) Implement the **GetAllProducts** method using the following code.

```
GetAllProducts(): Product[] {  
    return this.products;  
};
```


- j) Implement the **GetProduct** method using the following code.

```
GetProduct(id: number): Product {
    let products: Product[] = this.products.filter(product => product.Id === id);
    let product: Product = products[0];
    return product;
};
```

- k) Implement the **AddProduct** method using the following code.

```
AddProduct(product: Product): void {
    let ids: number[] = this.products.map(p => p.Id);
    let newId = Math.max(...ids) + 1;
    product.Id = newId;
    this.products.push(product);
};
```

- l) Implement the **DeleteProduct** method using the following code.

```
DeleteProduct(id: number): void {
    let index = this.products.map(product => product.Id).indexOf(id);
    this.products.splice(index, 1);
};
```

- m) Implement the **UpdateProduct** method using the following code.

```
UpdateProduct(product: Product): void {
    let index = this.products.map(product => product.Id).indexOf(product.Id);
    this.products[index] = product;
};
```

- n) Implement the **GetProductCategories** method using the following code.

```
GetProductCategories(): string[] {
    return ["Action Figures", "Arts and Crafts", "Remote Control"];
};
```

- o) Implement the **GetProductsByCategory** method using the following code.

```
GetProductsByCategory(category: string): Product[] {
    return this.products.filter(product => product.Category === category);
}
```

You have now finished implementing the **InMemoryProductDataService** class. The next step is to create an instance of this class and to register it with Angular as a named Angular service using the name **ProductDataService**.

- p) After the closing curly brace for the **InMemoryProductDataService** class definition, add the following code to register an instance of the **InMemoryProductDataService** class as an Angular service named **ProductDataService**.

```
angular.module('myApp').service('ProductDataService', InMemoryProductDataService);
```

- q) Make sure you have added this line of code after the **InMemoryProductDataService** class definition but before the closing curly brace of the **myApp** module as shown in the following screenshot.

```
module myApp {

    export class InMemoryProductDataService implements IProductDataService...

    angular.module('myApp').service('ProductDataService', InMemoryProductDataService);

}
```

- r) Save your changes to **services.ts**.

Now it is time to modify the **ProductsController** class to use the **ProductDataService** to read and write product data.

5. Examine the current definition of the **ProductsController** class inside **controllers.ts**.

```
class ProductsController {
    static $inject: Array<string> = [];
    constructor() { }
}

app.controller('productsController', ProductsController);
```

6. Update the **ProductsController** class using the following code.

- a) Modify the static **\$inject** field to inject the built-in Angular **\$location** service and your custom **ProductDataService** service.

```
static $inject: Array<string> = ['$location', 'ProductDataService'];
```

- b) Underneath the static **\$inject** field, add two new fields named **products** and **productCategories** using the following code.

```
static $inject: Array<string> = ['$location', 'ProductDataService'];
products: Product[];
productCategories: string[];
```

- c) Modify the **constructor** parameter list to include the **\$location** service and the **ProductDataService**.

```
constructor(private $location: ng.ILocationService,
             private ProductDataService: IProductDataService) {
}
```

- d) Modify the **constructor** body to initialize the **products** field and the **productCategories** field.

```
constructor(private $location: ng.ILocationService,
             private ProductDataService: IProductDataService) {
    this.products = ProductDataService.GetAllProducts();
    this.productCategories = ProductDataService.GetProductCategories();
}
```

- e) Underneath the **constructor**, add a new method named **deleteProduct** using the following code.

```
deleteProduct(id: number) {
    this.ProductDataService.DeleteProduct(id);
    this.$location.path("/products");
}
```

- f) At this point, your **ProductsController** class definition should match the following code.

```
class ProductsController {

    static $inject: Array<string> = ['$location', 'ProductDataService'];
    products: Product[];
    productCategories: string[];

    constructor(private $location: ng.ILocationService,
                 private ProductDataService: IProductDataService) {
        this.products = ProductDataService.GetAllProducts();
        this.productCategories = ProductDataService.GetProductCategories();
    }

    deleteProduct(id: number) {
        this.ProductDataService.DeleteProduct(id);
        this.$location.path("/products");
    }
}

app.controller('productsController', ProductsController);
```

Now that you have implemented the **ProductsController** class, you will now add three more controller classes to the **controllers.ts** source file named **AddProductController**, **ViewProductController** and **EditProductController**.

- Underneath the code that registers **productsController**, add the following code with the **AddProductController** class definition along with a line of code to register **addProductController**.

```
class AddProductController {
  static $inject: Array<string> = ['$location', 'ProductDataService'];
  product: Product = new Product();
  productCategories: string[];
  constructor(private $location: ng.ILocationService,
               private ProductDataService: IProductDataService) {
    this.productCategories = ProductDataService.GetProductCategories();
  }
  addProduct() {
    this.ProductDataService.AddProduct(this.product);
    this.$location.path("/products");
  }
}

app.controller('addProductController', AddProductController);
```

The classes named **ViewProductController** and **EditProductController** will be designed as parameterized controllers where the **id** of a product will be passed at the end of the route URL. You will now create an interface to define the route parameter named **id**.

- Underneath the code that registers **addProductController**, add a new interface named **IProductRouteParams**.

```
interface IProductRouteParams extends ng.route.IRouteParamsService {
  id: string;
}
```

- Underneath the interface, add the **ViewProductController** class definition and code to register **viewProductController**.

```
class ViewProductController {
  static $inject: Array<string> = ['$routeParams', 'ProductDataService'];
  product: Product;
  constructor($routeParams: IProductRouteParams,
              ProductDataService: IProductDataService) {
    var id: number = parseInt($routeParams.id);
    this.product = ProductDataService.GetProduct(id);
  }
}

app.controller('viewProductController', ViewProductController);
```

- Finally, add the **EditProductController** class definition and code to register **viewProductController**.

```
class EditProductController {
  static $inject: Array<string> = ['$routeParams', '$location', 'ProductDataService'];
  product: Product;
  productCategories: string[];
  constructor(private $routeParams: IProductRouteParams,
               private $location: ng.ILocationService,
               private ProductDataService: IProductDataService) {
    let id: number = parseInt($routeParams.id);
    this.product = ProductDataService.GetProduct(id);
    this.productCategories = ProductDataService.GetProductCategories();
  }
  updateProduct() {
    this.ProductDataService.UpdateProduct(this.product);
    this.$location.path("/products");
  }
}

app.controller('editProductController', EditProductController);
```

- Save and close **controllers.ts**.

Now that you have added three new controller classes, you must update the code inside **app.ts** to initialize the application's route map with a new route for each of the three new controllers.

12. Modify **app.ts** to add three new routes for the three new controllers.
 - a) Open **app.ts** in an editor window if it is not already open.
 - b) Place your cursor after the code which calls **when** to register the **products** route.



```
$routeProvider
    .when("/", {
        templateUrl: 'App/views/home.html',
        controller: "homeController",
        controllerAs: "vm"
    })
    .when("/products", {
        templateUrl: 'App/views/products.html',
        controller: "productsController",
        controllerAs: "vm"
    })
    .when("/products/showcase", {
        templateUrl: 'App/views/productsShowcase.html',
        controller: "productShowcaseController",
        controllerAs: "vm"
    })
    .otherwise({ redirectTo: "/" });
```

- c) Copy and paste in the following code to register three new routes for the three new controllers.

```
.when("/products/add", {
    templateUrl: 'App/views/productsAdd.html',
    controller: "addProductController",
    controllerAs: "vm"
})
.when("/products/view/:id", {
    templateUrl: 'App/views/productsView.html',
    controller: "viewProductController",
    controllerAs: "vm"
})
.when("/products/edit/:id", {
    templateUrl: 'App/views/productsEdit.html',
    controller: "editProductController",
    controllerAs: "vm"
})
```

- d) The code in **app.ts** to register routes should now match the following screenshot.



```
.when("/products", {
    templateUrl: 'App/views/products.html',
    controller: "productsController",
    controllerAs: "vm"
})
.when("/products/add", {
    templateUrl: 'App/views/productsAdd.html',
    controller: "addProductController",
    controllerAs: "vm"
})
.when("/products/view/:id", {
    templateUrl: 'App/views/productsView.html',
    controller: "viewProductController",
    controllerAs: "vm"
})
.when("/products/edit/:id", {
    templateUrl: 'App/views/productsEdit.html',
    controller: "editProductController",
    controllerAs: "vm"
})
.when("/products/showcase", {
    templateUrl: 'App/views/productsShowcase.html',
    controller: "productShowcaseController",
    controllerAs: "vm"
})
.otherwise({ redirectTo: "/" });
```

Inspect the parameterized routes defined for the two Angular controllers named **viewProductController** and **editProductController**. A parameterized route is defined with a colon (:) and then a named parameter. For example, the route for **viewProductController** is defined as **/products/view/:id**. That means this route has been defined with a parameter named **id**.

13. Update **index.html** to add script links for **product.js** and **services.js**.

- Open **index.html** in a code editor window if it is not already open.
- Add two new script links for **product.js** and **services.js**.

```
<!-- Add script links to internal JS files -->
<script src="App/app.js"></script>
<script src="App/models/product.js"></script>
<script src="App/services/services.js"></script>
<script src="App/controllers/controllers.js"></script>
```

- Save your changes and close **index.html**.

What about script link to **interfaces.js**? It turns out that is unnecessary. TypeScript interface definitions are completely erased when the TypeScript code is compiled into JavaScript. If you examine **interfaces.js**, you will see that it is an empty file.

14. Implement the view template file name **products.html**.

- Using Windows Explorer, locate the file named **products.html.txt** in the **Students** folder at the following location.

C:\Student\Modules\04_AngularWebApps\Lab\StarterFiles\products.html.txt

- Double-click on **products.html.txt** to open this file in Notepad.
- Copy the contents of **products.html.txt** to the Windows clipboard.
- Return to Visual Studio and place the cursor inside the editor windows for **products.html**.
- Paste the content of the Windows clipboard into **products.html**.

```
<p>
  <a href="/products/add" class="btn btn-primary" >Create Product</a>
</p>

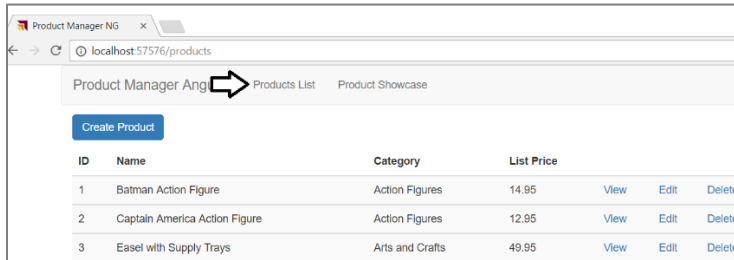
<table id="products-table" class="table table-striped table-hover table-responsive ">
  <thead>
    <tr>
      <th>ID</th>
      <th>Name</th>
      <th class="hidden-sm hidden-xs">Category</th>
      <th class="hidden-sm hidden-xs">List Price</th>
      <th>&nbsp;</th>
      <th>&nbsp;</th>
      <th>&nbsp;</th>
    </tr>
  </thead>
  <tbody ng-repeat="product in vm.products">
    <tr>
      <td>{{product.Id}}</td>
      <td>{{product.Name}}</td>
      <td class="hidden-sm hidden-xs">{{product.Category}}</td>
      <td class="hidden-sm hidden-xs">{{product.ListPrice }}</td>
      <td><a href="/products/view/{{product.Id}}" class="navbar-link" >view</a></td>
      <td><a href="/products/edit/{{product.Id}}" class="navbar-link" >edit</a></td>
      <td><a href="" ng-click="vm.deleteProduct(product.Id)" class="navbar-link" >Delete</a></td>
    </tr>
  </tbody>
</table>
```

Take a moment to review the HTML code you just pasted into **products.html**. First, you can see that there is a link button at the top that can be used to redirect the user to **/products/add** in order to add a new product to the products list. Next, examine how this view template generates an HTML table using product data acquired through the view model. The **tbody** tag in the middle of the table layout contains an **ng-repeat** directive which enumerates through the **vm.products** collection and generate a separate **<tr>** element to create a row in the table for each product. Since the **<tr>** element serves as a template, the **product** variable defined in the **ng-repeat** directive can be used to retrieve product properties for the current product such as its **Id**, **Name**, **Category** and **ListPrice** simply by using the syntax **{{product.Name}}**.

- Save and close **products.html**.

15. Test out the **ProductManagerAngular1** project using the Visual Studio Debugger

- Press the **{F5}** key to start up the project in the Visual Studio debugger.
- When the project starts, click the **Product List** link to navigate to the products controller.
- The view for the products controller should appear with an HTML table of products and match the following screenshot.



- Test out the **Delete** link on one or two of the products in the list.

ID	Name	Category	List Price			
1	Batman Action Figure	Action Figures	\$14.95	View	Edit	Delete ←
2	Captain America Action Figure	Action Figures	\$12.95	View	Edit	Delete

- Confirm that clicking the **Delete** link removes that product from this list.

ID	Name	Category	List Price			
2	Captain America Action Figure	Action Figures	\$12.95	View	Edit	Delete
3	Easel with Supply Trays	Arts and Crafts	\$49.95	View	Edit	Delete

After deleting a few products, you can refresh the page to restore the products list back to its original state. Now that's something you can't do in a real-world application. You just gotta love demoware!

16. Use the built-in Angular **currency** filter to format product list price.

- Inspect the list price of the product named **Twitter Follower Action Figure**. You can see the list price is displayed as **1** instead of **\$1.00** as you would expect for a currency value.

5	Green Stomper Bully	Remote Control	24.95	View
6	Indy Race Car	Remote Control	19.95	View
7	Twitter Follower Action Figure	Action Figures	1 ←	View
8	Sandpiper Prop Plane	Remote Control	24.95	View

- Leave the application running in the browser and return to Visual Studio.
- Inside **products.html**, add the built-in Angular filter named **currency** to format **product.ListPrice**.

```
<tr>
  <td>{{product.Id}}</td>
  <td>{{product.Name}}</td>
  <td class="hidden-sm hidden-xs">{{product.Category}}</td>
  <td class="hidden-sm hidden-xs">{{product.ListPrice | currency }}</td>
  <td><a href="/products/view/{{product.Id}}" class="navbar-link">View</a></td>
  <td><a href="/products/edit/{{product.Id}}" class="navbar-link">Edit</a></td>
</tr>
```

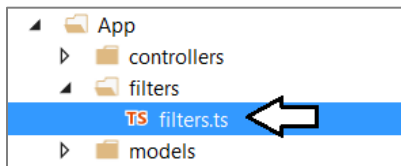
- Save your changes to **products.html**.

- e) Return to the browser and refresh the current page.
- f) You should see that the list price for each product now has currency formatting.

5	Green Stomper Bully	Remote Control	\$24.95	View
6	Indy Race Car	Remote Control	\$19.95	View
7	Twitter Follower Action Figure	Action Figures	\$1.00	View
8	Sandpiper Prop Plane	Remote Control	\$24.95	View

If you can't see the updated formatting, you may have to empty the browser cache to load the updated version of **products.html**.

- g) Close the browser, return to Visual Studio and stop the debugger.
17. Create a custom Angular filter named **listPrice** to format product list prices with a custom format.
- a) Add a new folder inside the **App** folder named **filters**.
 - b) Inside the **filters** folder, create a new TypeScript file named **filters.ts**.



- c) Inside **filters.ts**, add the following code to define and register a custom Angular filter named **listPrice**.

```
module myApp {
  class AppFilters {
    static $inject: Array<string> = ['$filter'];
    public static listPriceFilter($filter) {
      return (price: number) => {
        let formattedNumber = $filter('number')(price, 2);
        return "$" + formattedNumber + " USD";
      }
    }
  }

  angular.module("myApp").filter("listPrice", AppFilters.listPriceFilter);
}
```

This custom filter named **listPrice** leverages the built-in Angular filter named **number** to format the list price with two significant digits after the decimal. This custom filter also appends the **\$** before the number and **USD** after the number so values appear as **\$1.00 USD**.

- d) Save and close **filters.ts**.
- e) Open **index.html** and add a new script link for **filters.ts**.

```
<!-- Add script links to internal JS files -->
<script src="App/app.js"></script>
<script src="App/models/product.js"></script>
<script src="App/services/services.js"></script>
<script src="App/filters/filters.js"></script>
<script src="App/controllers/controllers.js"></script>
```

- f) Return to **products.html** and replace the built-in **currency** filter with your new custom filter named **listPrice**.

```
<td>{{product.Name}}</td>
<td class="hidden-sm hidden-xs">{{product.Category}}</td>
<td class="hidden-sm hidden-xs">{{product.ListPrice | listPrice}}</td>
<td><a href="/products/view/{{product.Id}}" class="navbar-link">View</a></td>
```

- g) Save your changes and close **products.html**.

18. Test out the **ProductManagerAngular1** project using the Visual Studio debugger

- Press the **{F5}** key to start up the project in the Visual Studio debugger.
- When the project starts, click the **Product List** link to navigate to the products controller.
- The product list prices should now be formatted with the **listPrice** filter and they should match the following screenshot.

ID	Name	Category	List Price
1	Batman Action Figure	Action Figures	\$14.95 USD
2	Captain America Action Figure	Action Figures	\$12.95 USD
3	Easel with Supply Trays	Arts and Crafts	\$49.95 USD

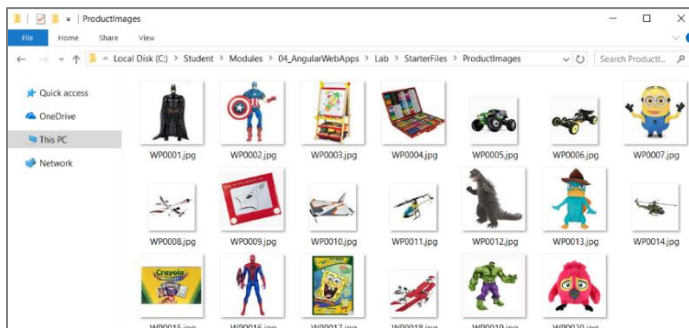
- Close the browser, return to Visual Studio and stop the debugger.

19. Add a set of product images into the **Content** folder.

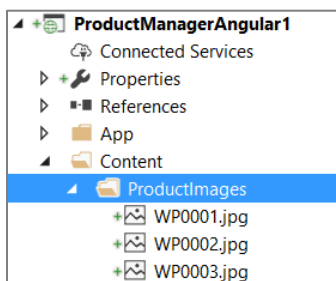
- Using Windows Explorer, locate and inspect the **ProductImages** folder inside the **Students** folder at the following location.

C:\Student\Modules\04_AngularWebApps\Lab\StarterFiles\ProductImages

- You should see that **ProductImages** folder contains a set of JPG files that contain product images.

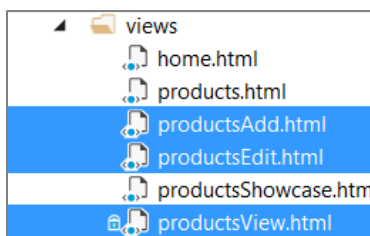


- Copy the **ProductImages** folder and the images inside into the **Content** folder of the **ProductManagerAngular1** project.



20. Add new view template files into the **App/views** folder.

- Add three new view template files named **productsAdd.html**, for **productsEdit.html** and **productsView.html**.



21. Implement the view file named **productsView.html**.

- a) Using Windows Explorer, locate the file named **productsView.html.txt** in the **Students** folder at the following location.

```
C:\Student\Modules\04_AngularWebApps\Lab\StarterFiles\productsView.html.txt
```

- b) Double-click on **productsView.html.txt** to open this file in Notepad.
c) Copy the contents of **productsView.html.txt** to the Windows clipboard.
d) Return to Visual Studio and place the cursor inside the editor window for **productsView.html**.
e) Paste the content of the Windows clipboard into **productsView.html**.

```
<h3>View Product</h3>

<div>
  <dl class="dl-horizontal">

    <dt>Name</dt>
    <dd ng-bind="vm.product.Name"></dd>

    <dt>Category</dt>
    <dd ng-bind="vm.product.Category"></dd>

    <dt>List Price</dt>
    <dd ng-bind="vm.product.ListPrice | listPrice"></dd>

    <dt>Description</dt>
    <dd ng-bind="vm.product.Description"></dd>

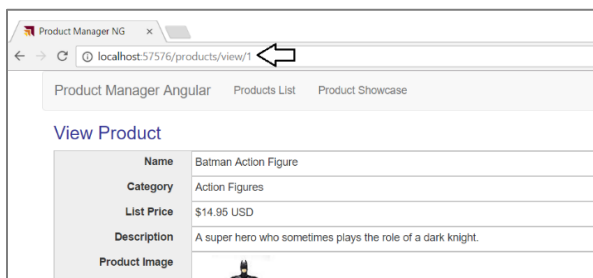
    <dt>Product Image</dt>
    <dd>
      
    </dd>
  </dl>
</div>
<p>
  <a href="/products" class="btn btn-link" >Back to products list</a> |
  <a href="/products/edit/{vm.product.Id}" class="btn btn-link">Edit this product</a>
</p>
```

Take a moment and review this HTML code to see how the layout is created to view the data for a single product.

- f) Save and close **productsView.html**.

22. Test out the **ProductManagerAngular1** project using the Visual Studio debugger

- a) Press the **{F5}** key to start up the project in the Visual Studio debugger.
b) When the project starts, click the **Product List** link to navigate to the products controller.
c) Click the **View** link for the product with the name of **Batman Action Figure**.
d) You should be redirected to the route **/products/view/1** and you should see a view that matches the following screenshot.



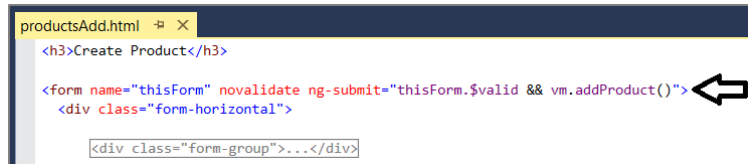
- e) Close the browser, return to Visual Studio and stop the debugger.

23. Implement the view file named **productsAdd.html**.

- a) Using Windows Explorer, locate the file named **productsAdd.html.txt** in the **Students** folder at the following location.

C:\Student\Modules\04_AngularWebApps\Lab\StarterFiles\productsAdd.html.txt

- b) Double-click on **productsAdd.html.txt** to open this file in Notepad.
c) Copy the contents of **productsAdd.html.txt** to the Windows clipboard.
d) Return to Visual Studio and place the cursor inside the editor windows for **productsAdd.html**.
e) Paste the content of the Windows clipboard into **productsAdd.html**.
f) Inspect the HTML opening HTML **form** tag at the top of **productsAdd.html**.



```
<h3>Create Product</h3>

<form name="thisForm" novalidate ng-submit="thisForm.$valid && vm.addProduct()">
  <div class="form-horizontal">
    <div class="form-group">...</div>
```

You can see that the form has been given a name of **thisForm**. Note that the **ng-submit** directive has been configured to only submit the form if all the form input values are valid. Also the HTML5 **novalidate** attribute is used to tell the browser not to generate error messages for invalid input values. When you use the **novalidate** attribute, you are then responsible for generating error and remedy messages for invalid input on the part of the user.

- g) Move down inside the code in **productsAdd.html** and inspect the **form-group** div for **product.Name**.

```
<div class="form-group">
  <label class="control-label col-md-2" for="Name">Name:</label>
  <div class="col-md-10">
    <input id="Name" name="Name" type="text" ng-model="vm.product.Name" required class="form-control text-box single-line" />
    <div class="text-danger" ng-show="thisForm.Name.$invalid && (thisForm.Name.$dirty || thisForm.$submitted)">
      <span ng-show="thisForm.Name.$error.required">Name is a required field</span>
    </div>
  </div>
</div>
```

You can see the input element named **Name** carries the **ng-model** attribute which binds it to the product name from the underlying view model. The **Name** input element has also been defined with the HTML5 **required** attribute which means the user must type in a value for **Name** and cannot leave it blank. Now look below the **Name** input element and inspect what's inside the div that has a class of **text-danger**. This code demonstrates a typical approach for using the Angular framework support for form validation.

- h) Move down inside the code in **productsAdd.html** and inspect the **form-group** div for **product.ListPrice**.

```
<div class="form-group">
  <label class="control-label col-md-2" for="ListPrice">List Price:</label>
  <div class="col-md-10">
    <input id="ListPrice" name="ListPrice" type="number" required min="1" max="10000" step="any"
      ng-model="vm.product.ListPrice" class="form-control" />
    <div class="text-danger" ng-show="thisForm.ListPrice.$invalid && (thisForm.ListPrice.$dirty || thisForm.$submitted)">
      <span ng-show="thisForm.ListPrice.$error.required">List price is a required field</span>
      <span ng-show="thisForm.ListPrice.$error.number">List price must be a number</span>
      <span ng-show="thisForm.ListPrice.$error.min || thisForm.ListPrice.$error.max">
        List price must be between $1 and $10,000
      </span>
    </div>
  </div>
</div>
```

You can see the input element named **ListPrice** carries the **ng-model** attribute which binds it to the product list price from the underlying view model. The **ListPrice** input element has also been defined with **required** attribute and the **number** attribute which means the user must type in a value that is numeric. The **min** and **max** attribute have also been added to define a range for the value of a list price. Now look below the **ListPrice** input element and inspect what's inside the div that has a class of **text-danger**. This code looks for three different possible validation errors and provides three different error messages.

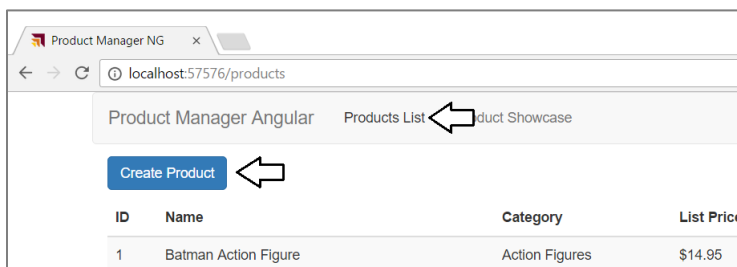
- i) Inspect the **button** element at the bottom of **productsAdd.html** with a **type** of **submit** and a caption of **Save**.

```
<div class="form-group">
  <div class="col-md-offset-2 col-md-10">
    <button type="submit" class="btn btn-primary">Save</button>
    <a href="/products" class="btn btn-default" accesskeyrole="button">Cancel</a>
  </div>
</div>
</div>
</form>
```

- j) Save and close of **productsAdd.html**.

24. Test out the **ProductManagerAngular1** project using the Visual Studio Debugger

- Press the **{F5}** key to start up the project in the Visual Studio debugger.
- When the project starts, click the **Product List** link to navigate to the products controller.
- Click the **Create Product** button to test the user experience for creating a new product.



- d) You should be redirected to the **Create Product** view as shown in the following screenshot.

Create Product

Name:

Category:

List Price:

Description:

Product Image:

- e) Before entering any product data, click the **Save** button to see the behavior of the form validation logic. Once the user has attempted to submit the form, an error message will show for each invalid form input value.

Create Product

Name: Name is a required field

Category: Category is a required field

List Price: List price is a required field

Description: Description is a required field

Product Image:

- f) Type in a product Name of Spiderman Action Figure and select Action Figure for the Category.

- g) Type in a **List Price** of **0.5** and tab out of the **List Price** textbox. You should see a new error message on the price amount.

The screenshot shows the 'Create Product' form with the following fields: Name (Spiderman Action Figure), Category (Action Figures), List Price (0.5), and Description (A classic superhero who is also quite the swinger). The List Price field is highlighted with a red border, and a red error message is displayed below it: 'List price must be between \$1 and \$10,000'.

- h) Fill in the required data in the **Create Product** form as shown in the following screenshot and then click **Save**.

The screenshot shows the 'Create Product' form with the following fields: Name (Spiderman Action Figure), Category (Action Figures), List Price (14.95), Description (A classic superhero who is also quite the swinger), and Product Image (WP0016.jpg). The Save button is highlighted with a red border.

Make sure to use a **Product Image** value of **WP0016.jpg** because that is the image to use for the **Spiderman Action Figure**.

- i) You should now be redirected back to the products list view.
j) Look at the bottom of the products list and verify you can see the new **Spiderman Action Figure** product.

13	Perry the Platypus Action Figure	Action Figures	\$21.95	View	Edit	Delete
14	Seal Team 6 Helicopter	Remote Control	\$59.95	View	Edit	Delete
15	Crayola Crayon Set	Arts and Crafts	\$2.49	View	Edit	Delete
16	Spiderman Action Figure	Action Figures	\$14.95	View	Edit	Delete

- k) Click the **View** link for the **Spiderman Action Figure** product to see the product details.

The screenshot shows the 'View Product' page for the Spiderman Action Figure. The page displays the product details: Name (Spiderman Action Figure), Category (Action Figures), List Price (\$14.95 USD), Description (A classic superhero who is also quite the swinger), and Product Image (WP0016.jpg). The Product Image is a small image of Spiderman. At the bottom, there are links for 'Back to products list' and 'Edit this product'.

- l) Close the browser, return to Visual Studio and stop the debugger.

Now that you have implemented the view to add products, you will move on to the last step of implementing the view to edit products.

25. Implement the view file named **productsEdit.html**.

- a) Using Windows Explorer, locate the file named **productsEdit.html.txt** in the **Students** folder at the following location.

C:\Student\Modules\04_AngularWebApps\Lab\StarterFiles\productsEdit.html.txt

- b) Double-click on **productsEdit.html.txt** to open this file in Notepad.
- c) Copy the contents of **productsEdit.html.txt** to the Windows clipboard.
- d) Return to Visual Studio and place the cursor inside the editor windows for **productsEdit.html**.
- e) Paste the content of the Windows clipboard into **productsEdit.html**.
- f) Quickly review the code inside **productsEdit.html**.

Note that the HTML code in **productsEdit.html** is almost identical to the HTML code in **productsAdd.html**. The Angular form validation logic you create can often be directly reused between an add view and an edit view.

26. Test out the **ProductManagerAngular1** project using the Visual Studio debugger
 - a) Press the **{F5}** key to start up the project in the Visual Studio debugger.
 - b) When the project starts, click the **Product List** link to navigate to the products controller.
 - c) The **Batman Action Figure** product should currently have a list price of **\$14.95**.

ID	Name	Category	List Price			
1	Batman Action Figure	Action Figures	14.95	View	Edit	Delete

- d) Click the **Edit** link for the **Batman Action Figure** product to test the user experience of editing an existing product.
- e) Update the product **List Price** from **14.95** to **19.95** and then click **Update** to save your changes.

Update Product

Name:

Batman Action Figure

Category:

Action Figures

List Price:

19.95

Description:

A super hero who sometimes plays the role of a dark knight.

Product Image:

WP0001.jpg

Update

- f) Verify that you can see the update to the **List Price** for the **Batman Action Figure** in the products list.

ID	Name	Category	List Price	
1	Batman Action Figure	Action Figures	\$19.95	View

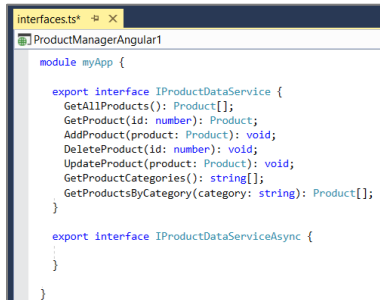
- g) Close the browser, return to Visual Studio and stop the debugger.

You have now implemented a complete Angular application with full CRUD functionality. In the next exercise, you will begin to modify code within your controllers to use asynchronous execution to better reflect real-world applications which must read and write data across a network connection. You will begin the next exercise by creating a new interface named **IProductDataServiceAsync** which provides the same capabilities as the **IProductDataService** interface but it does so using asynchronous methods instead of synchronous methods. Then you will import code to implement a class named **AsyncInMemoryProductDataService** that implements the **IProductDataServiceAsync**. At the end of the next exercise, you will rewrite the TypeScript code inside your controller classes to use the **AsyncInMemoryProductDataService** class instead of the **InMemoryProductDataService** class. This give you experience writing code in a controller to execute methods on a service that only supports asynchronous behavior. However, there is good news in that you will not have to make any more modifications to your view templates because each of your controllers will continue to create the exact same view model after being updated to use an asynchronous interface.

Exercise 4: Create and Implement an Asynchronous Interface

In this exercise you will create a new interface named **IProductDataServiceAsync** which provides an asynchronous version of the **IProductDataService** interface. Next, you will import code to implement a new class named **AsyncInMemoryProductDataService** that implements the **IProductDataServiceAsync**. After that, you will update the code in your controller classes to use the **AsyncInMemoryProductDataService** class instead of the **InMemoryProductDataService** class.

1. Create the **IProductDataServiceAsync** interface.
 - a) Open the **interfaces.ts** file in the **Apps/models** folder.
 - b) Under the **IProductDataService** interface, add a new interface definition named **IProductDataServiceAsync**.



```
export interface IProductDataService {
    GetAllProducts(): Product[];
    GetProduct(id: number): Product;
    AddProduct(product: Product): void;
    DeleteProduct(id: number): void;
    UpdateProduct(product: Product): void;
    GetProductCategories(): string[];
    GetProductsByCategory(category: string): Product[];
}

export interface IProductDataServiceAsync {
}
```

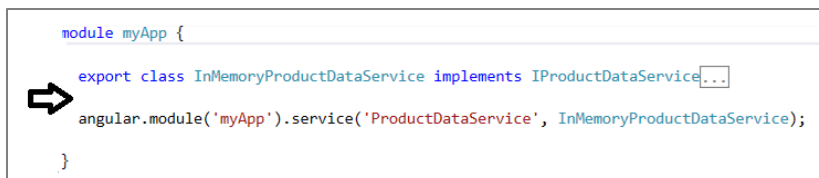
- c) Create the definition for the **IProductDataServiceAsync** interface using the following code.

```
export interface IProductDataServiceAsync {
    GetAllProductsAsync(): ng.IPromise<Product[]>;
    GetProductAsync(id: number): ng.IPromise<Product>;
    AddProductAsync(product: Product): ng.IPromise<void>;
    DeleteProductAsync(id: number): ng.IPromise<void>;
    UpdateProductAsync(product: Product): ng.IPromise<void>;
    GetProductCategoriesAsync(): ng.IPromise<string[]>;
    GetProductsByCategoryAsync(category: string): ng.IPromise<Product[]>;
}
```

- d) Save your changes and close **interfaces.ts**.
2. Import code to implement **AsyncInMemoryProductDataService**.
 - a) Using Windows Explorer, locate the file named **AsyncInMemoryProductDataService.cs.txt** in the **Students** folder.

C:\Student\Modules\04_AngularwebApps\Lab\StarterFiles\AsyncInMemoryProductDataService.cs.txt

- b) Double-click on **AsyncInMemoryProductDataService.cs.txt** to open this file in Notepad.
 - c) Copy the contents of **AsyncInMemoryProductDataService.cs.txt** to the Windows clipboard.
 - d) Return to Visual Studio and place the cursor inside the editor window for **services.ts**.
 - e) Paste the content of the Windows clipboard to the existing class definition.



```
module myApp {

    export class InMemoryProductDataService implements IProductDataService {
    }

    angular.module('myApp').service('ProductDataService', InMemoryProductDataService);
}
```

You do not have to modify any code in the **AsyncInMemoryProductDataService** class. You just need to copy-and-paste the completed class definition. If you are curious, you can optionally examine the code inside the **AsyncInMemoryProductDataService** class to see how the code is implemented. However, the more important aspect of this lab involves you modifying the code inside your controller classes to program against the **IProductDataServiceAsync** interface.

- a) After you have pasted the code, there should now be an implementation of the **AsyncInMemoryProductDataService** class below the implementation of the **InMemoryProductDataService** class.

```
module myApp {  
  
    export class InMemoryProductDataService implements IProductDataService{...}  
  
    export class AsyncInMemoryProductDataService implements IProductDataServiceAsync{...}  
  
    angular.module('myApp').service('ProductDataService', InMemoryProductDataService);  
  
}
```

- b) Locate the bottom line of code in **services.ts** that calls the **service** method to register the **ProductDataService** service.
c) Update the second parameter from **InMemoryProductDataService** to **AsyncInMemoryProductDataService**.

```
module myApp {  
  
    export class InMemoryProductDataService implements IProductDataService{...}  
  
    export class AsyncInMemoryProductDataService implements IProductDataServiceAsync{...}  
  
    angular.module('myApp').service('ProductDataService', AsyncInMemoryProductDataService);  
  
}
```

- d) Save your changes and close **services.ts**.

Now the only remaining work is to update your controller class inside **controllers.ts** to use the **IProductDataServiceAsync** interface.

3. Update all controller code inside **controllers.ts** to use **IProductDataServiceAsync** instead of **IProductDataService**.
a) Open **controllers.ts** in a code editor window if it's not already open.
b) Locate the **constructor** inside the **ProductsController** class.
c) Currently, the **ProductDataService** constructor parameter is defined using the **IProductDataService** interface.

```
class ProductsController {  
    static $inject: Array<string> = ['$location', 'ProductDataService'];  
    products: Product[];  
    productCategories: string[];  
    // add constructor  
    constructor(private $location: ng.ILocationService,  
                private ProductDataService: IProductDataService){  
        this.products = ProductDataService.GetAllProducts();  
        this.productCategories = ProductDataService.GetProductCategories();  
    }  
}
```

- d) Update the **ProductDataService** constructor parameter type from **IProductDataService** to **IProductDataServiceAsync**.

```
constructor(private $location: ng.ILocationService,  
            private ProductDataService: IProductDataServiceAsync){  
    this.products = ProductDataService.GetAllProducts();  
    this.productCategories = ProductDataService.GetProductCategories();  
}  
// add delete user action  
deleteProduct(id: number) {  
    this.ProductDataService.DeleteProduct(id);  
    this.$location.path("/products");  
}
```

You will see compile errors after you change the parameter type from type from **IProductDataService** to **IProductDataServiceAsync**.

- e) Currently, the code in the constructor is still using the synchronous methods **GetAllProducts** and **GetProductCategories**.

```
constructor(private $location: ng.ILocationService,  
            private ProductDataService: IProductDataServiceAsync){  
    this.products = ProductDataService.GetAllProducts();  
    this.productCategories = ProductDataService.GetProductCategories();  
}
```

- f) Update the constructor to call the asynchronous methods **GetAllProductsAsync** and **GetProductCategoriesAsync** using the following code which calls the **then** method and supplies an anonymous callback function for each asynchronous call.

```
constructor(private $location: ng.ILocationService,
             private ProductDataService: IProductDataServiceAsync) {

    ProductDataService.GetAllProductsAsync()
        .then((result: Product[]) => {
            this.products = result;
        });

    ProductDataService.GetProductCategoriesAsync()
        .then((result: string[]) => {
            this.productCategories = result;
        });
}
```

- g) Update the **deleteProduct** method to call **DeleteProductAsync** instead of **DeleteProduct..**

```
deleteProduct(id: number) {
    this.ProductDataService.DeleteProductAsync(id)
        .then(() => {
            this.$location.path("/products");
        });
}
```

You have now updated the **ProductsController** class. Next, you must update the other controller classes inside **controllers.ts** named **AddProductController**, **ViewProductController** and **EditProductController** so that they use the **IProductDataServiceAsync** interface and its asynchronous methods instead of the **IProductDataService** interface. Once you update these three controller classes, you will be ready to test out your work.

- h) Modify the **AddProductController** class to program against **IProductDataServiceAsync** using the following code.

```
class AddProductController {
    static $inject: Array<string> = ['$location', 'ProductDataService'];
    product: Product = new Product();
    productCategories: string[];

    constructor(private $location: ng.ILocationService,
                private ProductDataService: IProductDataServiceAsync) {
        ProductDataService.GetProductCategoriesAsync()
            .then((result: string[]) => {
                this.productCategories = result;
            });
    }

    addProduct() {
        this.ProductDataService.AddProductAsync(this.product)
            .then(() => {
                this.$location.path("/products");
            });
    }
}
```

- i) Modify the **ViewProductController** class to program against **IProductDataServiceAsync** using the following code.

```
class ViewProductController {
    static $inject: Array<string> = ['$routeParams', 'ProductDataService'];
    product: Product;
    constructor($routeParams: IProductRouteParams, ProductDataService: IProductDataServiceAsync) {
        var id: number = parseInt($routeParams.id);
        ProductDataService.GetProductAsync(id)
            .then((result: Product) => {
                this.product = result;
            });
    }
}
```


- j) Modify the **EditProductController** class to program against **IProductDataServiceAsync** using the following code.

```
class EditProductController {
    static $inject: Array<string> = ['$routeParams', '$location', 'ProductDataService'];
    product: Product;
    productCategories: string[];
    constructor(private $routeParams: IProductRouteParams,
                private $location: ng.ILocationService,
                private ProductDataService: IProductDataServiceAsync) {

        var id: number = parseInt($routeParams.id);
        ProductDataService.GetProductAsync(id)
            .then((result: Product) => {
                this.product = result;
            });

        ProductDataService.GetProductCategoriesAsync()
            .then((result: string[]) => {
                this.productCategories = result;
            });
    }

    updateProduct() {
        this.ProductDataService.UpdateProductAsync(this.product)
            .then(() => { this.$location.path("/products"); });
    }
}
```

- k) Save your changes to **controllers.ts**.
4. Test out the **ProductManagerAngular1** project using the Visual Studio Debugger
- Press the **{F5}** key to start up the project in the Visual Studio debugger.
 - Test out the application and ensure you can still create, view, edit and delete products.

Product Manager Angular

Products List

Product Showcase

Create Product

ID	Name	Category	List Price			
1	Batman Action Figure	Arts and Crafts	\$14.95	View	Edit	Delete
2	Captain America Action Figure	Action Figures	\$12.95	View	Edit	Delete

- c) Close the browser, return to Visual Studio and stop the debugger.

Running the updated version of your application in the Visual Studio debugger might seem a bit anticlimactic. That's because the application should look and behave the exactly as it did at the end of the last exercise. However, you know conceptually quite a bit has changed under the covers. Your controller classes are now programmed against an asynchronous service.

Exercise 5: Create a Custom Navigation Menu using an Angular Component

In this exercise you will update the **ProductShowcaseController** class and its view template named **productsShowcase.html** to provide an enhanced user experience to examining the catalog of products. Along the way, you will create a new Angular component to implement navigation menu which allows users to search through the product catalog by product category.

- Update the **ProductShowcaseController** class to provide a better view model.
 - Open **controllers.ts** in a code editor window if it is not already open.
 - Examine the current implementation of the **ProductShowcaseController** class.

```
class ProductShowcaseController {
    static $inject: Array<string> = [];
    constructor() { }
}
```

- c) Update the initializer for the **\$inject** field to include the **\$location** service and the **ProductDataService** service.

```
static $inject: Array<string> = ['$location', 'ProductDataService'];
```

- d) In between the **\$inject** field and the constructor, add a new field named **product** based on an array of **Product** instances.

```
class ProductShowcaseController {  
  static $inject: Array<string> = ['$location', 'ProductDataService'];  
  products: Product[];  
  constructor() { }  
}
```

- e) Update the parameter list of the constructor with one parameter named **\$location** based on the type **ng.ILocationService** and a second parameter named **ProductDataService** based on the type **IPProductDataServiceAsync**.

```
class ProductShowcaseController {  
  static $inject: Array<string> = ['$location', 'ProductDataService'];  
  products: Product[];  
  constructor(private $location: ng.ILocationService,  
              private ProductDataService: IPProductDataServiceAsync) {  
    // constructor code goes here  
  }  
}
```

- f) Implement the constructor by calling **GetAllProductsAsync** to initialize the **products** field.

```
class ProductShowcaseController {  
  static $inject: Array<string> = ['$location', 'ProductDataService'];  
  products: Product[];  
  constructor(private $location: ng.ILocationService,  
              private ProductDataService: IPProductDataServiceAsync) {  
    // constructor code goes here  
    ProductDataService.GetAllProductsAsync()  
      .then((result: Product[]) => {  
        this.products = result;  
      });  
  }  
}
```

- g) Save your changes to **controllers.ts** but do not close it. You will return to this source file to make one more edit a little later.
2. Modify the **productsShowcase.html** view template with a new and improved HTML layout.
- a) Open the view template named **productsShowcase.html** from inside the **App/views** folder.
- b) Delete the HTML inside **productsShowcase.html** and replace it with the following HTML code.

```
<div class="row">  
  <div class="container">  
    <div class="row row-offcanvas row-offcanvas-left">  
  
      <!-- sidebar nav -->  
  
      <!-- main area -->  
  
    </div>  
  </div>  
</div>
```

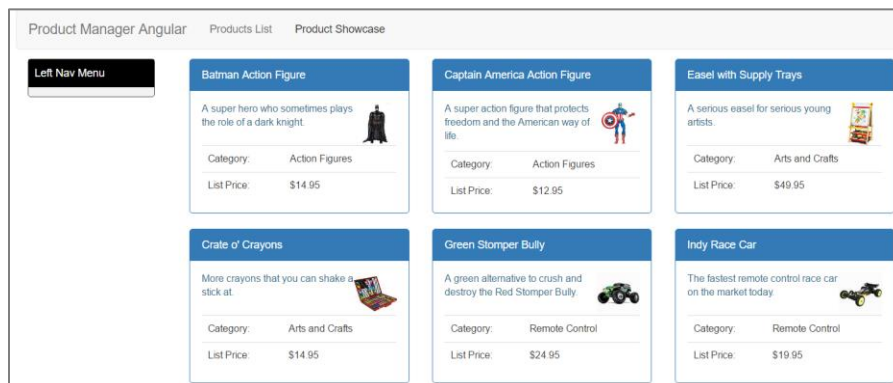
- c) Locate the **sidebar nav** comment.
- d) Copy and paste the following HTML code below the **sidebar nav** comment for a left navigation menu.

```
<!-- sidebar nav -->  
<div class="col-xs-6 col-sm-2 sidebar-offcanvas" id="sidebar" role="navigation">  
  <nav id="left-nav" class="navbar" role="navigation">  
    <div id="left-nav-title">Left Nav Menu</div>  
  </nav>  
</div>
```

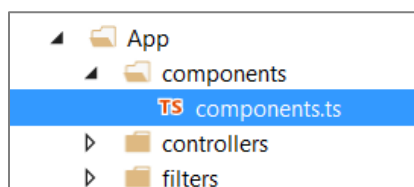
- e) Locate the **main area** comment.
- f) Copy and paste the following HTML code below the **main area** comment to create a view template which uses the **ng-repeat** directive to generate an inline-block **<div>** element for each product.

```
<!-- main area -->
<div class="col-xs-12 col-sm-10">
  <div class="col-lg-4 col-md-5 col-sm-6" ng-repeat="product in vm.products">
    <div class="panel panel-primary productPanel">
      <div class="panel-heading">{{product.Name}}</div>
      <div class="panel-body">
        
        <p class="text-info">{{product.Description}}</p>
        <div class="" style="clear: both;">
          <table class="table">
            <tr>
              <td>Category:</td>
              <td>{{product.Category}}</td>
            </tr>
            <tr>
              <td>List Price:</td>
              <td>{{product.ListPrice | currency }}</td>
            </tr>
          </table>
        </div>
      </div>
    </div>
  </div>
</div>
```

- g) Save your changes to **productsShowcase.html** but don't close it because you will be making one more edit in a bit.
3. Test out the **ProductManagerAngular1** project using the Visual Studio Debugger.
- a) Press the **{F5}** key to start up the project in the Visual Studio debugger.
 - b) Click on the **Product Showcase** link in the top navigation and examine how the product data is displayed.



- c) Close the browser, return to Visual Studio and stop the debugger.
4. Create an Angular component named **productNavigation** to serve as the left navigation menu.
- a) Inside the **App** folder, create a new folder named **components**.
 - b) Inside the **components** folder, create a new TypeScript source file named **components.ts**.



- c) Inside **components.ts**, begin by adding the following code to create a variable named **app** which references the Angular application and two starter class definitions for two classes named **ProductNavigationController** and **ProductNavigation**.

```
module myApp {  
    let app = angular.module("myApp");  
    class ProductNavigationController { }  
    class ProductNavigation { }  
}
```

Creating an Angular component involves two classes. One class is used to implement a controller class while the other class is used to define the component options. As you will see, the component options class will specify binding options, a controller class and a view template for the component.

- d) Update the **ProductNavigationController** class with the following code.

```
class ProductNavigationController {  
    static $inject: string[] = ['ProductDataService'];  
    public productCategories: string[];  
  
    constructor(private ProductDataService: IProductDataServiceAsync) {  
        // initialize view model inside $onInit and not inside constructor  
    };  
  
    public $onInit() {  
        this.ProductDataService.GetProductCategoriesAsync()  
            .then((result: string[]) => {  
                this.productCategories = result;  
            });  
    }  
}
```

- e) Modify to **ProductNavigation** class declaration to implement the **ng.IComponentOptions** interface.

```
class ProductNavigation implements ng.IComponentOptions
```

- f) Implement the **ProductNavigation** class using the following TypeScript code.

```
class ProductNavigation implements ng.IComponentOptions {  
  
    public bindings: { [binding: string]: string };  
    public controller: any;  
    public templateUrl: any;  
  
    constructor() {  
        this.bindings = {};  
        this.controller = ProductNavigationController;  
        this.templateUrl = '/App/components/productNavigation.html';  
    }  
}
```

- g) At the bottom of **components.ts**, add the following code to register the component with the angular application.

```
app.component("productNavigation", new ProductNavigation());
```

Take note that this component has been given a name of **productNavigation**. This name will be important when you decide to use this component by instantiating it inside a view template with an HTML tag. You can see that the name **productNavigation** has been created using camel case where the first letter is lowercase and there is a capital letter at the start of each word after the first word. For example, **productNavigation** has one capital letter (N) for the second word "Navigation". When you want to create an HTML tag for this component in a view template, you replace each capital letter with a hyphen and then the lowercase version of that letter. That means a component named **productNavigation** can be instantiated with an HTML tag of **<product-navigation />**.

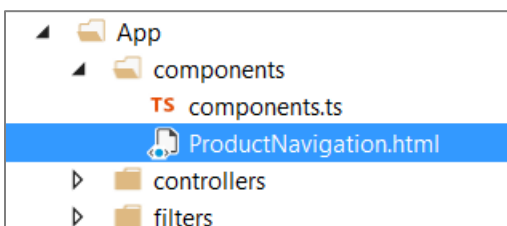
- h) When you are done editing **components.ts**, this source file should match the following code listing.

```
module myApp {  
  
    let app = angular.module("myApp");  
  
    class ProductNavigationController {  
        static $inject: string[] = ['ProductDataService'];  
        public productCategories: string[];  
  
        constructor(private ProductDataService: IProductDataServiceAsync) {  
            // initialize view model inside $onInit not in constructor  
        };  
  
        public $onInit() {  
            this.ProductDataService.GetProductCategoriesAsync()  
                .then((result: string[]) => {  
                    this.productCategories = result;  
                });  
        }  
    }  
  
    class ProductNavigation implements ng.IComponentOptions {  
  
        public bindings: { [binding: string]: string };  
        public controller: any;  
        public templateUrl: any;  
  
        constructor() {  
            this.bindings = {};  
            this.controller = ProductNavigationController;  
            this.templateUrl = '/App/components/productNavigation.html';  
            console.log("component constructor executing");  
        }  
    }  
  
    app.component("productNavigation", new ProductNavigation());  
}
```

- i) Save your changes and close **components.ts**.
5. Add a new script link for **components.ts**.
- a) Open **index.html** and add a new script link for **components.ts**.

```
<!-- Add script links to internal JS files -->  
<script src="App/app.js"></script>  
<script src="App/models/product.js"></script>  
<script src="App/services/services.js"></script>  
<script src="App/filters/filters.js"></script>  
<script src="App/components/components.js"></script>  
<script src="App/controllers/controllers.js"></script>
```

- b) Save your changes and close **index.html**.
6. Add a new view template for the **productNavigation** component.
- a) Inside the **App/components** folder, add a new HTML file named **ProductNavigation.html**.



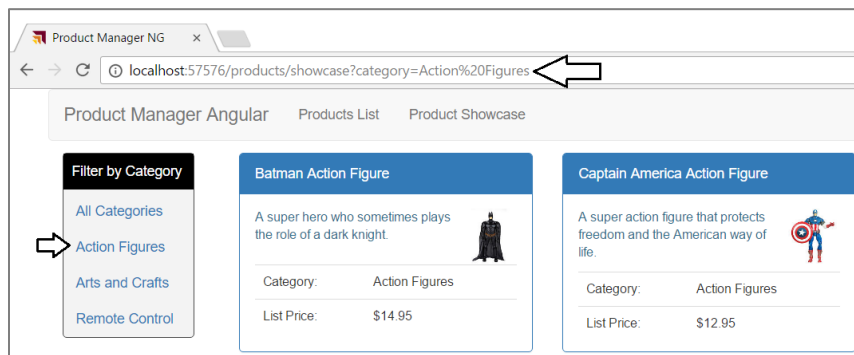
- b) Delete any code that existing within **ProductNavigation.html** and replace it with the following HTML code.

```
<nav id="left-nav" class="navbar" role="navigation">
  <div id="left-nav-title">Filter by Category</div>
  <ul class="nav navbar-nav">
    <li class="nav navbar-link"><a href="/products/showcase/">All Categories</a></li>
    <li class="nav navbar-link" ng-repeat="category in $ctrl.productCategories">
      <a href="/products/showcase/?category={{category}}">{{category}}</a>
    </li>
  </ul>
</nav>
```

- c) Save your changes and close **ProductNavigation.html**.
7. Modify the view template named **productsShowcase.html** to use the new component.
- a) Return to the editor window for **productsShowcase.html**.
- b) Replace the HTML underneath the **sidebar nav** comment with the following code to instantiate the **productNavigation** component using a **<product-navigation />** tag.

```
<!-- sidebar nav -->
<div class="col-xs-6 col-sm-2 sidebar-offcanvas" id="sidebar" role="navigation">
  <product-navigation />
</div>
```

- c) Save your changes and close **productsShowcase.html**.
8. Test out the **ProductManagerAngular1** project using the Visual Studio debugger
- a) Press the **{F5}** key to start up the project in the Visual Studio debugger.
- b) Click on the **Product Showcase** link in the top navigation and examine the left navigation menu.
- c) Click on the **Action Figures** link in the left navigation and observe how to adds a query string parameter named **category** to the end of the route with the HTML-encoded value of **Action%20Figures**.



The left navigation menu currently updates the route with a query string parameter for the selected **category**. However, it does not yet actually change the set of visible products. Now in the final step of this lab you will modify the **ProductShowcaseController** class to filter the set of products when it determines the route contains the **category** query string parameter.

9. Update the constructor of the **ProductShowcaseController** class to filter products using the **category** query string parameter.
- a) Return to **controllers.ts** and move down to the class definition of the **ProductShowcaseController** class.
- b) Examine the current implementation of the constructor.

```
constructor(private $location: ng.ILocationService,
             private ProductDataService: IProductDataServiceAsync) {
  // constructor code
  ProductDataService.GetAllProductsAsync()
    .then((result: Product[]) => {
      this.products = result;
    });
}
```

- c) Add a line of code at the beginning of the constructor to retrieve the **category** query string parameter.

```
let categoryFilter: string = $location.search().category;
```

- d) Update the code that follows to call **GetAllProductsAsync** when the **categoryFilter** is undefined. However, the code should call **GetProductsByCategoryAsync** when **categoryFilter** has a valid value.

```
if (categoryFilter === undefined) {  
    ProductDataService.GetAllProductsAsync()  
        .then((result: Product[]) => {  
            this.products = result;  
        });  
}  
else {  
    ProductDataService.GetProductsByCategoryAsync(categoryFilter)  
        .then((result: Product[]) => {  
            this.products = result;  
        });  
}
```

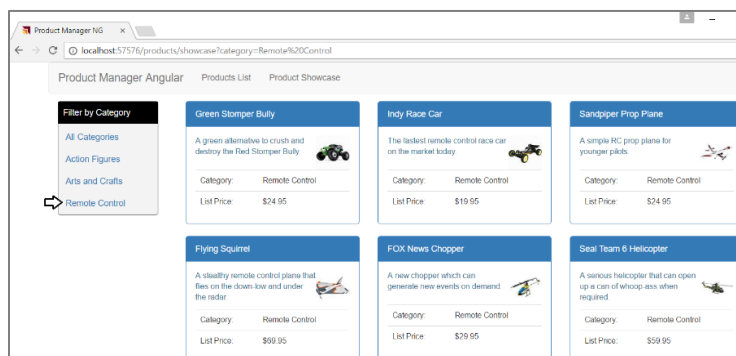
- e) When you are done, the constructor of the **ProductShowcaseController** should match the following code listing.

```
constructor(private $location: ng.ILocationService,  
            private ProductDataService: IProductDataServiceAsync) {  
  
    let categoryFilter: string = $location.search().category;  
  
    if (categoryFilter === undefined) {  
        ProductDataService.GetAllProductsAsync()  
            .then((result: Product[]) => {  
                this.products = result;  
            });  
    }  
    else {  
        ProductDataService.GetProductsByCategoryAsync(categoryFilter)  
            .then((result: Product[]) => {  
                this.products = result;  
            });  
    }  
}
```

- f) Save your changes and close **controllers.ts**.

10. Test out the **ProductManagerAngular1** project using the Visual Studio debugger

- a) Press the **{F5}** key to start up the project in the Visual Studio debugger.
b) Click on the **Product Showcase** link in the top navigation and test out the left navigation menu.



Congratulations. You have now reached the end of this lab.

If you have more time and you are up for a challenge, see how long it takes you to deploy the **ProductManagerAngular1** project to a new Azure web app with a URL ending with **azurewebsites.net** to make it accessible to anyone on the Internet.