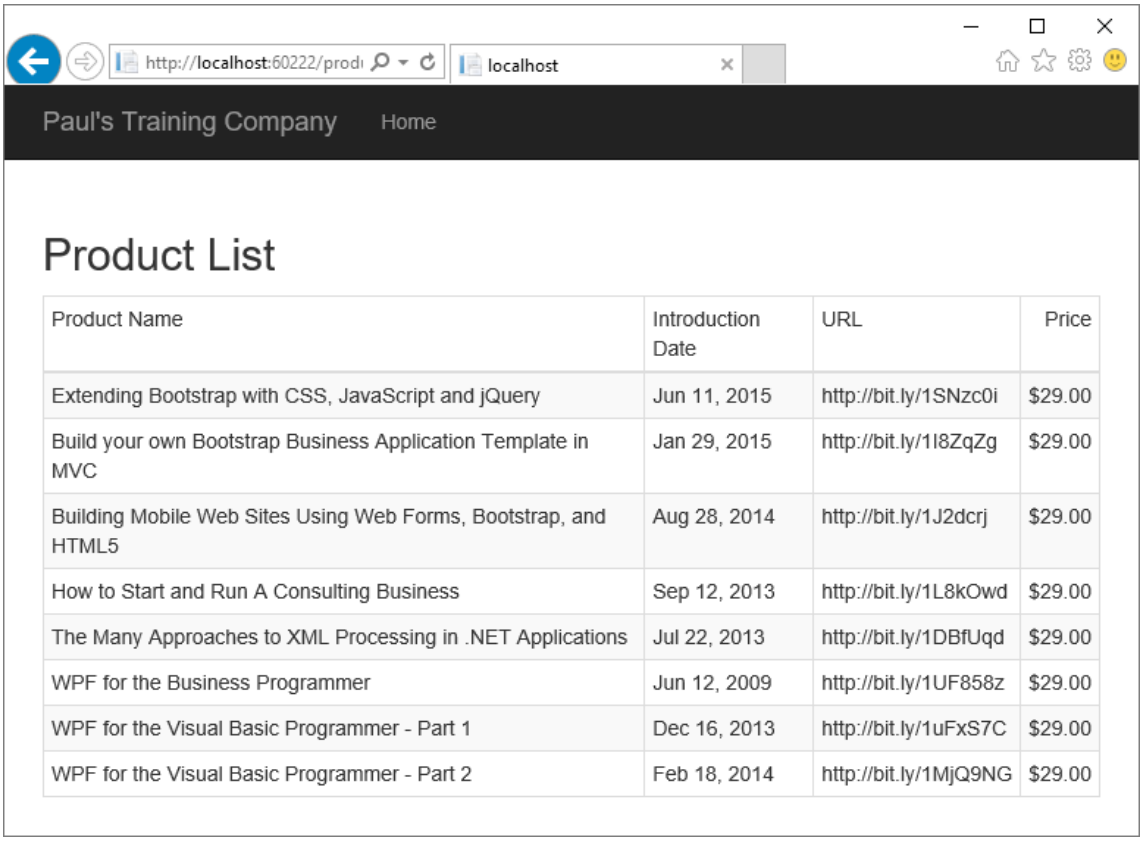
Add Angular to MVC – Part 2

In the first part of this blog series, you added Angular 2 to an MVC application using Visual Studio. In this blog post, you will learn how to add a Web API that can be called from an Angular service. You will modify the Global.asax to automatically convert pascal-cased properties in C# classes into camel-cased TypeScript properties. You will build an Angular service, learn to inject it into a component, then call the service to retrieve product data. Finally, you will take the returned data and build an HTML table. For this post, I am assuming you are a Microsoft Visual Studio developer and are familiar with MVC, Angular, C#, and the Web API.



Product Name	Introduction Date	URL	Price
Extending Bootstrap with CSS, JavaScript and jQuery	Jun 11, 2015	http://bit.ly/1SNzc0i	$29.00
Build your own Bootstrap Business Application Template in MVC	Jan 29, 2015	http://bit.ly/1I8ZqZg	$29.00
Building Mobile Web Sites Using Web Forms, Bootstrap, and HTML5	Aug 28, 2014	http://bit.ly/1J2dcrj	$29.00
How to Start and Run A Consulting Business	Sep 12, 2013	http://bit.ly/1L8kOwd	$29.00
The Many Approaches to XML Processing in .NET Applications	Jul 22, 2013	http://bit.ly/1DBfUqd	$29.00
WPF for the Business Programmer	Jun 12, 2009	http://bit.ly/1UF858z	$29.00
WPF for the Visual Basic Programmer - Part 1	Dec 16, 2013	http://bit.ly/1uFxS7C	$29.00
WPF for the Visual Basic Programmer - Part 2	Feb 18, 2014	http://bit.ly/1MjQ9NG	$29.00

Figure 1: The product list page written using Angular

Open up the project you created in the last blog post and follow along with the steps outlined in this one to create the final project.

Create Web API

To retrieve data from an Angular application, or any client-side technology, you create a Web API. Right mouse click on the \Controllers folder and select **Add | Web API Controller Class (v2.1)** from the menu. Set the name to **ProductApiController** and click the OK button.

Delete all the code within this new controller class as you are going to write your Web API methods using a more updated approach than what is generated by Visual Studio. Type in the code listed in the code snippet below to create a method that retrieves all products from the mock data returned from the ProductViewModel class.

```
public IHttpActionResult Get() {
  IHttpActionResult ret;
  ProductViewModel vm = new ProductViewModel();

  vm.LoadProducts();
  if (vm.Products.Count() > 0) {
    ret = Ok(vm.Products);
  }
  else {
    ret = NotFound();
  }

  return ret;
}
```

Add WebApiConfig Class

If you did not have the Web API in your MVC application before, then you need to add a WebApiConfig class to your project. Look in the the \App_Start folder and see if the WebApiConfig.cs class exists. If not, right mouse click on the \App_Start folder and add a new class. Set the name to WebApiConfig and click the OK button. Make the class file look like the following:

```
using System.Web.Http;

namespace PTC
{
  public static class WebApiConfig
  {
    public static void Register(HttpConfiguration config) {
      // Web API configuration and services

      // Web API routes
      config.MapHttpAttributeRoutes();

      config.Routes.MapHttpRoute(
          name: "DefaultApi",
          routeTemplate: "api/{controller}/{id}",
          defaults: new { id = RouteParameter.Optional }
      );
    }
  }
}
```

This code adds a new route template to the routes accepted by this MVC application. Any call that starts with “api” is assumed to be a Web API call and thus the ASP.NET engine knows to look for an class with the specified controller name, and that inherits from the ApiController class.

Fix the Global.asax

If you open the \Models\Product class you see a standard C# class definition. As you can imagine, you are going to have a similar class on the client-side that maps all properties one-to-one from this C# class to the TypeScript class. However, the C# class uses pascal-casing of properties, while our TypeScript class is going to use camel-casing. You can add code to the Application_Start method in the Global.asax file to automatically convert properties from pascal-case to camel-case. Open the Global.asax file and at the top of the file add the three using statements shown in this code snippet.

```
using Newtonsoft.Json.Serialization;
using System.Web.Http;
using System.Net.Http.Formatting;
```

Add a single line of code to the Application_Start method to call the WebApiConfig.Register() method you just wrote. Place this line of code before you call the RouteConfig.RegisterRoutes() method.

```
protected void Application_Start() {
  AreaRegistration.RegisterAllAreas();
  FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

  // Add Web API
  GlobalConfiguration.Configure(WebApiConfig.Register);

  RouteConfig.RegisterRoutes(RouteTable.Routes);
  BundleConfig.RegisterBundles(BundleTable.Bundles);
}
```

Next, add code after all the registration calls to perform the pascal-case to camel-case conversion for you.

```
// Get Global Configuration
HttpConfiguration config =
     GlobalConfiguration.Configuration;

// Convert to camelCase
var jsonFormatter = config.Formatters
  .OfType<JsonMediaTypeFormatter>()
   .FirstOrDefault();

jsonFormatter.SerializerSettings
  .ContractResolver = new
     CamelCasePropertyNamesContractResolver();
```

The above code selects the current GlobalConfiguration.Configuration property and assigns it a variable called *config*. The next line of code queries the Formatters collection and retrieves the first instance of a JsonMediaTypeFormatter object it finds. Finally, you create a new instance of a CamelCasePropertyNamesContractResolver and assign that to the formatter's ContractResolver property. This property controls how the JSON objects are formatted and sent to the client-side caller.

Build Client-Side Product List

Now that you have your server-side pieces in place, and have gotten Angular to work in your project, you can start building the HTML and classes to create a list of products. There are two new files you are going to create in this part of the article. One is a Product class and the other is a Product Service class.

Create Product Class

As you are going to retrieve a collection of Product classes from the Web API, you need a Product class written in TypeScript. Right mouse click on the \app\product folder and select **Add | TypeScript file** from the context-sensitive menu. Give this new file the name of **product.ts**. Add the following code in this file.

```
export class Product {
  productId: number;
  productName: string;
  introductionDate: Date;
  price: number;
  url: string;
  categoryId: number;
}
```

Notice that the structure of this class is exactly like the Product class that the Entity Framework generated. The only difference is the use of camel-case instead of pascal-case. But, as you remember we added code in the Global.asax to handle this conversion for us.

Create Product Service

We need an Angular product service to call the Web API controller you created earlier. This product service will retrieve all products. Right mouse click on the \app\products folder and select **New | TypeScript file** from the menu. Set the name to **product.service.ts** and click the OK button. Add the code shown in Listing 1.

```
import { Injectable } from '@angular/core';
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/catch';
import 'rxjs/add/observable/throw';

import { Product } from "./product";

@Injectable()
export class ProductService {
  private url = "api/productApi";

  constructor(private http: Http) {
  }

  getProducts(): Observable<Product[]> {
    return this.http.get(this.url)
      .map(this.extractData)
      .catch(this.handleErrors);
  }

  private extractData(res: Response) {
    let body = res.json();
    return body || {};
  }

  private handleErrors(error: any): Observable<any> {
    let errors: string[] = [];

    switch (error.status) {
      case 404: // Not Found
        errors.push("No Product Data Is Available.");
        break;

      case 500: // Internal Error
        errors.push(error.json().exceptionMessage);
        break;

      default:
        errors.push("Status: " + error.status
                    + " - Error Message: "
                    + error.statusText);
        break;
    };

    console.error('An error occurred', errors);

    return Observable.throw(errors);
  }
}
```

Listing 1: The product service retrieves a collection of products from your Web API.

Let's break down the code shown in Listing 1. You first import the various classes and functions you need for this service using the 'import' keyword.

```
import { Injectable } from '@angular/core';
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/catch';
import 'rxjs/add/observable/throw';

import { Product } from "./product";
```

The injectable class is needed to support the use of the @Injectable decorator. This decorator informs Angular that this class may be injected into the constructor of any component. The Http and Response classes are used to access the Web API and to retrieve the response from the Web API. Next is the Observable extension function supplied by RxJS or Reactive Extensions for JavaScript. This is a library that helps you transform, compose and query any type of data. In the ProductService class, you use it to take the result set of data you retrieve from the Web API and turn it into an observable collection of data. Next are a series of different reactive extensions to support the map, catch and throw functions you use in this class. The last import is the Product class you created. Since you want to return an array of product objects returned from the Web API, you must import this class.

Angular will inject the Http service into our ProductService class. Any class marked with the @Injectable decorator may be injected in any class by declaring it in the constructor.

```
constructor(private http: Http) {
}
```

The next two functions in our class are getProducts and extractData. The getProducts function calls our Web API using the get function of the http service. It creates an observable array of products using the map extension function. If data is retrieved, the 'map' function is called and passed a reference to the extractData function. You don't really need the extractData function, but I like having this function as it makes it easy for me to set a breakpoint in this function to see the data that has been returned. The extractData function is passed in the HTTP response object. Call the json function on this response object to retrieve the array of product data.

```
getProducts(): Observable<Product[]> {
  return this.http.get(this.url)
    .map(this.extractData)
    .catch(this.handleErrors);
}

private extractData(res: Response) {
  let body = res.json();
  return body || {};
}
```

The last function in this class is handleErrors.

```
private handleErrors(error: any): Observable<any> {
  let errors: string[] = [];

  switch (error.status) {
    case 404: // Not Found
      errors.push("No Product Data Is Available.");
      break;

    case 500: // Internal Error
      errors.push(error.json().exceptionMessage);
      break;

    default:
      errors.push("Status: " + error.status
                  + " - Error Message: "
                  + error.statusText);
      break;
  };

  console.error('An error occurred', errors);

  return Observable.throw(errors);
}
```

This function should be used to publish the error information. I am just going to output the error to the console window of the browser. You inform the calling code of the error by calling the throw function on the Observable class. The Observable returns an array of strings. For this sample, you only need a single error message to be returned, however, in other cases you will want multiple error messages. You will see examples of this when handling validation error messages.

Update the app.module.ts File

In order to inject the HTTP service in the ProductService class you need to import the HttpModule in the app.module.ts file. You then add this imported module to the @NgModule *imports* property. This makes it available for use in

any of your classes. Angular will take care of injecting this service whenever it sees a reference to Http as you saw in the constructor of your ProductService class.

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

@NgModule({
  imports: [BrowserModule, AppRoutingModule, HttpModule ]
  ...
})
```

Remember you marked your ProductService as an @Injectable class. This means you want Angular to automatically inject this service into any of your class constructors. To accomplish this, you also need to import the ProductService in your app.module.ts file and add this class to the *providers* property.

```
import { ProductService }
  from "./product/product.service";

@NgModule({
  imports: [BrowserModule, HttpModule ],
  declarations: [ AppComponent ],
  bootstrap: [AppComponent],
  providers: [ProductService]
})
```

Update Product List HTML Page

It is now finally time to build the components needed to display a list of products like the page shown in Figure 1. Open the \app\product\product-list.component.html you built earlier. Add the HTML shown in Listing 2.

```
<h2>Product List</h2>

<div class="row"
     *ngIf="products && products.length">
  <div class="col-xs-12">
    <div class="table-responsive">
      <table class="table table-bordered
                    table-condensed table-striped">
        <thead>
          <tr>
            <td>Product Name</td>
            <td>Introduction Date</td>
            <td>URL</td>
            <td class="text-right">Price</td>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let product of products">
            <td>
              {{product.productName}}
            </td>
            <td>
              {{product.introductionDate | date }}
            </td>
            <td>
              {{product.url}}
            </td>
            <td class="text-right">
              {{product.price | currency:'USD':true }}
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
</div>

<div class="row"
     *ngIf="messages && messages.length">
  <div class="col-xs-12">
    <div class="alert alert-warning">
      <ul>
        <li *ngFor="let msg of messages">{{msg}}</li>
      </ul>
    </div>
  </div>
</div>
```

Listing 2: Build the HTML table using the Angular *ngFor directive.

In the HTML that builds the product table, use the *ngIf directive to test if there any products. If there are, then you build an HTML table using the *ngFor directive to iterate over a collection named *products* in the ProductListComponent class. Each time through the loop a product object is

assigned to a local variable named *product*. Use this *product* variable to retrieve each property of the Product class and display the value of each in each cell of the table.

At the end of the HTML is another element which displays any error messages returned from the ProductListComponent class. This message is set if an error occurs in the retrieval of the product data from the Web API.

Update Product List Component

From the HTML in Listing 2 you know that your ProductListComponent class (Listing 3) must expose two properties to the HTML page; *products* and *message*. This class must be injected with the ProductService class from which you can request the product data. This class must also have some exception handling in case the Web API throws an exception or does not return data. Open the product-list.component.ts under the \app\product folder. Modify this class to look like the code contained in Listing 3.

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

import { Product } from "./product";
import { ProductService } from "./product.service";

@Component({
  moduleId: module.id,
  templateUrl: "./product-list.component.html"
})
export class ProductListComponent implements OnInit {
  constructor(private productService: ProductService) {
  }

  ngOnInit(){
    this.getProducts();
  }

  // Public properties
  products: Product[] = [];
  messages: string[] = [];

  private getProducts(){
    this.productService.getProducts()
      .subscribe(products => this.products = products,
       errors => this.handleErrors(errors));
  }

  private handleErrors(errors: any) {
    for (let msg of errors) {
      this.messages.push(msg);
    }
  }
}
```

Listing 3: The product listing component.

Let's look at each piece of the ProductListComponent class and learn why you need each line of code, and what each does. The first step is to import the various classes you use within this class. The following four imports are used in this class.

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

import { Product } from "./product";
import { ProductService } from "./product.service";
```

You can see the Component import is needed for the @Component decorator. The *templateUrl* property in the decorator points to the product-list.component.html page you just created.

The OnInit import is needed to support the ngOnInit lifecycle event which is raised when the Angular engine creates an instance of this class. We use the

ngOnInit function to call the getProducts function to retrieve the product data from the Web API.

The last two imports are easy to understand; we need the Product class because we are expecting to fill an array of Product objects from our call to the Web API. The ProductService class is used to call that Web API.

The constructor for this class receives the ProductService class from Angular injection and assigns that instance to the private property named *productService*.

```
constructor(private productService: ProductService) {
}
```

Two properties are declared in this class; *products* and *messages*. The *products* property is an array of Product objects. Initialize this property to an empty array by using the = []; syntax after the declaration of this property. The messages property is a string array used to display any error messages on the page.

```
// Public properties
products: Product[] = [];
messages: string[] = [];
```

The getProducts function calls the getProducts function from the ProductService class. This function returns an observable array of products from the map function, so use the subscribe function to retrieve the product array and assign it to the *products* property in this class. If an error occurs when calling the Web API, the function in the second parameter to the subscribe function is called. Take the error object passed back from the ProductService and pass that to the handleErrors function in this class.

```
getProducts() {
  this.productService.getProducts()
    .subscribe(products => this.products = products,
     errors => this.handleErrors(errors));
}
```

The handleErrors function simply loops through the list of error messages sent by the Observable.throw() in the Product service and adds each message to the *messages* array. This array is bound to an unordered list on the product-list.component.html page.

```
handleErrors(errors: any) {
  for (let msg of errors) {
    this.messages.push(msg);
  }
}
```

Update ProductController

Now that you are retrieving data from the Web API you created, you no longer need the code in the Get() or the Post() methods in your ProductController class. Open the ProductController class and delete the Post() method. Modify the Get() method to look like the following.

```
public ActionResult Product() {
    return View();
}
```

Summary

In these last two blog posts you added Angular into an MVC application. You learned to use Angular routing to route from MVC to Angular. You created a Web API on your server to return data to an Angular service. You then displayed a list of the data returned on an HTML page. In the next blog post you learn to add a new product.

Sample Code

You can download the code for this sample at www.pdsa.com/downloads. Choose the category “PDSA Blogs”, then locate the sample **Add Angular to MVC – Part 2**. NOTE: After downloading the sample, you will need to right mouse click on the package.json file and select the menu “Restore Packages”.