Call the Web API from Angular

This is the 3rd blog post in our series on Angular. The first part entitled **Get Started with Angular** shows you how to add Angular to a project. The second part, **Build Lists of Data Using Angular**, showed you how to build lists of data using hard-coded arrays of object literals. You should read those two blog entries first if you are not familiar with adding Angular to a project, don't know what a module or a controller is, or want to understand basic data binding.

In this blog post, you will take the product page with the HTML table and call a Web API to retrieve product data. The data returned from the Web API builds the HTML table of products using the **ng-repeat** directive.

To prepare for calling a Web API from your Angular controller, you need to do a little setup. For purposes of this blog post, you will be doing the following things.

1. Build Product classes to hold product data
2. Add a Web API Controller to your Project
3. Ensure you have configured ASP.NET to use the Web API
4. Call the Web API from Angular

Build Product Classes

Instead of using JavaScript object literals to populate an HTML table, you want to retrieve a set of objects from some data store on a server. Web API will take one of your classes and serialize it into a JSON object. To start, create a class named **TrainingProduct** that contains all of the properties to represent a product object. Create the class shown below in your web project.

```
public class TrainingProduct
{
  public int ProductId { get; set; }
  public string ProductName { get; set; }
  public DateTime IntroductionDate { get; set; }
  public string Url { get; set; }
  public decimal Price { get; set; }
  public int CategoryId { get; set; }
}
```

Next, build a class named **TrainingProductManager** that is responsible for building a collection of TrainingProduct objects. You could use the Entity Framework, Haystack or any other data layer you want to retrieve data from a Product table in a database. To keep things simple in this post, I am just going to create some mock data as shown in the code below.

```
public class TrainingProductManager
{
  public List<TrainingProduct> Get()
  {
    List<TrainingProduct> ret = new List<TrainingProduct>();

    // TODO: Add your own data access method here
    ret = CreateMockData();

    return ret;
  }

  protected List<TrainingProduct> CreateMockData()
  {
    List<TrainingProduct> ret = new List<TrainingProduct>();

    ret.Add(new TrainingProduct()
    {
      ProductId = 1,
      ProductName = "Extending Bootstrap with CSS,
                    JavaScript and jQuery",
      IntroductionDate = Convert.ToDateTime("6/11/2015"),
      Url = "http://bit.ly/1SNzc0i",
      Price = Convert.ToDecimal(29.00),
      CategoryId = 1
    });

    ret.Add(new TrainingProduct()
    {
      ProductId = 2,
      ProductName = "Build your own Bootstrap Business
                     Application Template in MVC",
      IntroductionDate = Convert.ToDateTime("1/29/2015"),
      Url = "http://bit.ly/1I8ZqZg",
      Price = Convert.ToDecimal(29.00),
      CategoryId = 1
    });

    // MORE ENTRIES HERE
}
```

Once you have a generic List of TrainingProduct objects, it is this data that is returned from the Web API. As I stated previously, Web API will automatically serialize this generic List into an array of JSON objects and return that to your Angular controller.

Add Web API to your Project

Depending on how you built your project you may or may not have the appropriate DLLs and other necessary artifacts to support an ASP.NET Web API controller class. I am going to assume that you have not used a Web API in your project and show you how to add them to your project.

Add a new folder \Controllers-Api. While it is not necessary to add a new folder to contain your Web API controller classes, I like keeping them separate from my MVC controllers.

Right mouse click on your Visual Studio project and select **Add | New Folder** from the content-sensitive menu. Set the name to **Controllers-Api** and press the enter key.

Add a new controller by right-mouse clicking on the new folder you just added and select **Add | Web API Controller Class (v2.1)**. NOTE: If this option does not show up on your Add menu, select New Item…, drill into the **Web | Web API** folder on the dialog and select Web API Controller Class (v2.1). Set the name to **ProductController** and click the OK or the Add button.

Within the ProductController class locate the following method stub created for you by Visual Studio.

```
// GET api/<controller>
public IEnumerable<string> Get() {
  return new string[] { "value1", "value2" };
}
```

Replace this method with the following code. NOTE: You may need to include the Namespace in which you created the TrainingProduct and TrainingProductManager classes.

```
[HttpGet()]
public IHttpActionResult Get() {
  IHttpActionResult ret = null;
  TrainingProductManager mgr =
    new TrainingProductManager();
  List<TrainingProduct> list;

  // Get all Products
  list = mgr.Get();
  if (list.Count > 0) {
    ret = Ok(list);
  }
  else {
    ret = NotFound();
  }

  return ret;
}
```

In the code above you create an instance of the TrainingProductManager class as you use that to get a collection of TrainingProduct objects. The return value from your Web API should always be an instance of an IHttpActionResult. This message includes an HTTP status code such as 200 or 404 plus any data you wish to send. In this case you use the Ok() method that is part of the ApiController class. This method accepts any data you wish to send, in this case the list of products, and sets a 200 status code. This is where the Web API framework automatically serializes your data into JSON. If there are no products to return use the NotFound() method to return a 404 status code to inform the front end that no data was found.

Configure Web API Routes

If you did not have any Web API features in your project before, you may need to register your intent to use the Web API with ASP.NET. This means that you have to configure the route that MVC should follow to locate any Web API controller so MVC knows they are different from a page controller. To do this, create a different prefix for calling your API controllers such as "api". The easiest method to accomplish this registration is to create a class called WebApiConfig. Add this class to the \App_Start folder in your project. Type the following code into this new class. *NOTE: Check to make sure this class does not already exist in your \App_Start folder.*

```
using System.Web.Http;

public static void Register(
       HttpConfiguration config)
{
  // Web API routes
  config.MapHttpAttributeRoutes();

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

The above code first configures the ASP.NET runtime to allow for Web API calls by calling the MapHttpAttributeRoutes method. Next, it adds a new HttpRoute and sets up a new template with the “api” prefix. When you make a call to your Web API controllers you always prefix them with “api” to distinguish them from a normal MVC page controller. You can use any prefix you desire, but “api” is an industry convention.

Now that you have this class and method defined you need to call it from the Application_Start method in the Global.asax. Open the Global.asax file and add two new using statements. The first will be to the System.Web.HTTP namespace. This is needed to access the GlobalConfiguration class that is responsible for configuring the ASP.NET runtime with your new Web API template. The second namespace will the name of your project, followed by App_Start which is the folder name used when you create a class within that folder.

```
using System.Web.HTTP;
using [YOUR PROJECT NAME].App_Start;
```

Locate the Application_Start() method and add the line of code GlobalConfiguration.Configure() above the RouteConfig.RegisterRoutes() method call as shown in the code below.

```
protected void Application_Start() {

  AreaRegistration.RegisterAllAreas();
  FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

  GlobalConfiguration.Configure(WebApiConfig.Register);

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

Call Web API from your Controller

Now that you have all the classes in place for a Web API call, you are now ready to call it from your Angular code. Open up the productController.js file and modify the beginning of the function PTCController. Add one more core service that Angular provides, namely the data service that communicates with remote HTTP servers via a browser's built-in mechanism such as XMLHttpRequest or JSONP. The name of this core Angular service is passed to your controller as **$http**. Just like you did with your $scope variable, let's assign this to your own variable. Use the name **dataService** in this sample.

```
function PTCController($scope, $http) {
  var vm = $scope;
  var dataService = $http;
```

Locate where you declared the vm.products variable in the previous blog post and modify the variable to be an empty array instead of hard-coded object literals as shown in the code below.

```
// Expose a 'products' collection
vm.products = [];
```

In future posts I will cover how to deal with exceptions that come back from Web API calls, but for now, let's just create a function to handle any exceptions by grabbing the data.ExceptionMessage property that comes back from the Web API calls.

```
function handleException(error) {
  alert(error.data.ExceptionMessage);
}
```

Write a function called productList() to make the Web API call using the data service passed into our controller. The data service provided by Angular exposes several shortcut methods corresponding to the appropriate HTTP verbs, namely; get, post, put, delete and a few others. Since we are performing a GET to retrieve our product data from our Web API, call the get() method on the data service.

```
function productList() {
  dataService.get("/api/Product")
    .then(function (result) {
      vm.products = result.data;

    }, function (error) {
      handleException(error);
    });
}
```

Let's break down the various components of the get() method. Here is an overview of this function.

```
$http.get("URL TO WEB API")
     .then(success function (result){},
           error   function (error) {});
```

The parameter you pass to the get() method is the URL to your Web API controller. This method returns a "promise" which is nothing more than an object that takes care of making an asynchronous call for you. The chained method then() is used to inform the promise of what function to call if the Web API is successful and which function to call if an error is returned. In each case an object is passed back to you with some properties that you can query to see what happened. The objects are the same for both success and failure and contain the following properties:

data = Either a string or object containing data from your Web API call

status = An HTTP status code number such as 200 or 404

headers = A function to allow you to get the headers from the call

config = A configuration object that was used for sending the request

statusText = An HTTP status text returned from the call

Look at the productList() function you wrote and see that in the success function, you take the data property from the result parameter and assign that to your array named vm.products. By doing this, you allow the data binding of

Angular to take over and update any bound items on the page. In this case the ng-repeat attached to the <tr> on your HTML table causes a redrawing of your HTML table based on the new data in the array.

Now that you have the function productList() written, you need to call it when you enter your controller. Immediately after declaring all your variables on your scope, call the productList() function. Below is what the complete productController.js file should now look like.

```
(function () {
  'use strict';

  angular.module('ptcApp')
    .controller('PTCController', PTCController);

  function PTCController($scope, $http) {
    var vm = $scope;
    var dataService = $http;

    // Expose a 'product' object
    vm.product = {
      ProductName: 'Pluralsight Subscription'
    };
    // Create a list of categories
    vm.categories = [
      { CategoryName: 'Videos' },
      { CategoryName: 'Books' },
      { CategoryName: 'Articles' }
    ];
    vm.products = [];

    productList();

    function handleException(error) {
      alert(error.data.ExceptionMessage);
    }

    function productList() {
      dataService.get("/api/Product")
        .then(function (result) {
          vm.products = result.data;

        }, function (error) {
          handleException(error);
        });
    }
  }
})();
```

The last thing you need to do is to modify the building of the product table data. Let's add a couple more columns to our HTML table using the following code:

```
<tr ng-repeat="product in products">
  <td>{{product.ProductName}}</td>
  <td>{{product.IntroductionDate | date: mm/dd/yyyy }}</td>
  <td>{{product.Url}}</td>
  <td class="text-right">{{product.Price | currency: $ }}</td>
</tr>
```

In this new ng-repeat you are adding the IntroductionDate and Url to the table. Notice the use of the “date” filter that assists you in formatting the IntroductionDate data. Figure 1 shows you what the final HTML product table looks like after running this sample.

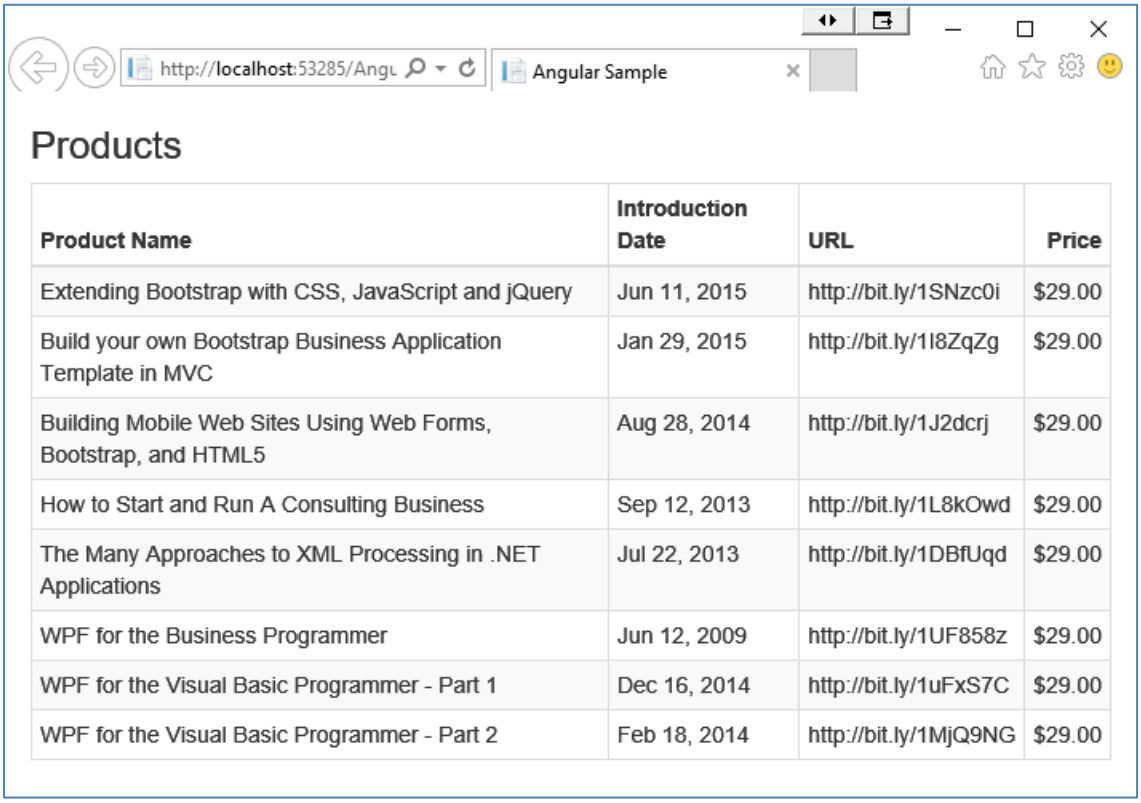

Figure 1: A list of products retrieved via the Web API.

Summary

The $http data service is built-in to Angular. All you need to do is to add an additional parameter in your controller function. It is recommended you assign the $http variable to your own variable name for flexibility. Also in this post you learned to configure your web project to make calls to a Web API.

Sample Code

You can download the code for this sample at www.pdsa.com/downloads. Choose the category “PDSA Blog”, then locate the sample **PDSA Blog Sample: Call the Web API from Angular**.