ASP.NET Web API 2

Beginner Guide

AMBILY K K

Contents

About the author

Ambily have been a Microsoft MVP in 2011 and a Microsoft Technology Evangelist**.** Her focus area includes .NET technologies, Windows 8, Team Foundation Server and HTML5/jQuery. She is also an author for Simple-Talk, CodeProject and dotnetfunda. Her blog is at http://ambilykk.com.

About this book

This short Book explains Web API design, concepts, features, and help page generation. This is a starter guide for those who want to quickly understand the basics of Web API. Topics covered in this book are:

- Implementing Web API
- Web API Client Implementations – ASP.NET MVC and jQuery
- Scaffolding with Web API – Entity Framework
- Routing in Web API
- Implementing Multiple Serialization Options
- Help Page Generation

Share your feedbacks, criticisms, and suggestions of improvements to author.ambily@outlook.com

Introduction

ASP.NET Web API is a framework for building HTTP services that can be accessed from various clients, such as browsers and mobile devices. ASP.NET Web API was introduced as part of ASP.NET MVC 4; however, it has its origins in WCF as WCF Web API. This new HTTP service model is simple to develop and contains common HTTP features, including Caching, Status Code, and so on.

In this short book, we will discuss Web API design, concepts, features, and compare Web API with WCF.

Implementing a Web API Project

Let's start our discussion with a sample Web API project. We will be using Visual Studio 2013 as our development environment. Our first step will be to create an ASP.NET MVC project based on the **Web API** template, as shown in Figure 1.

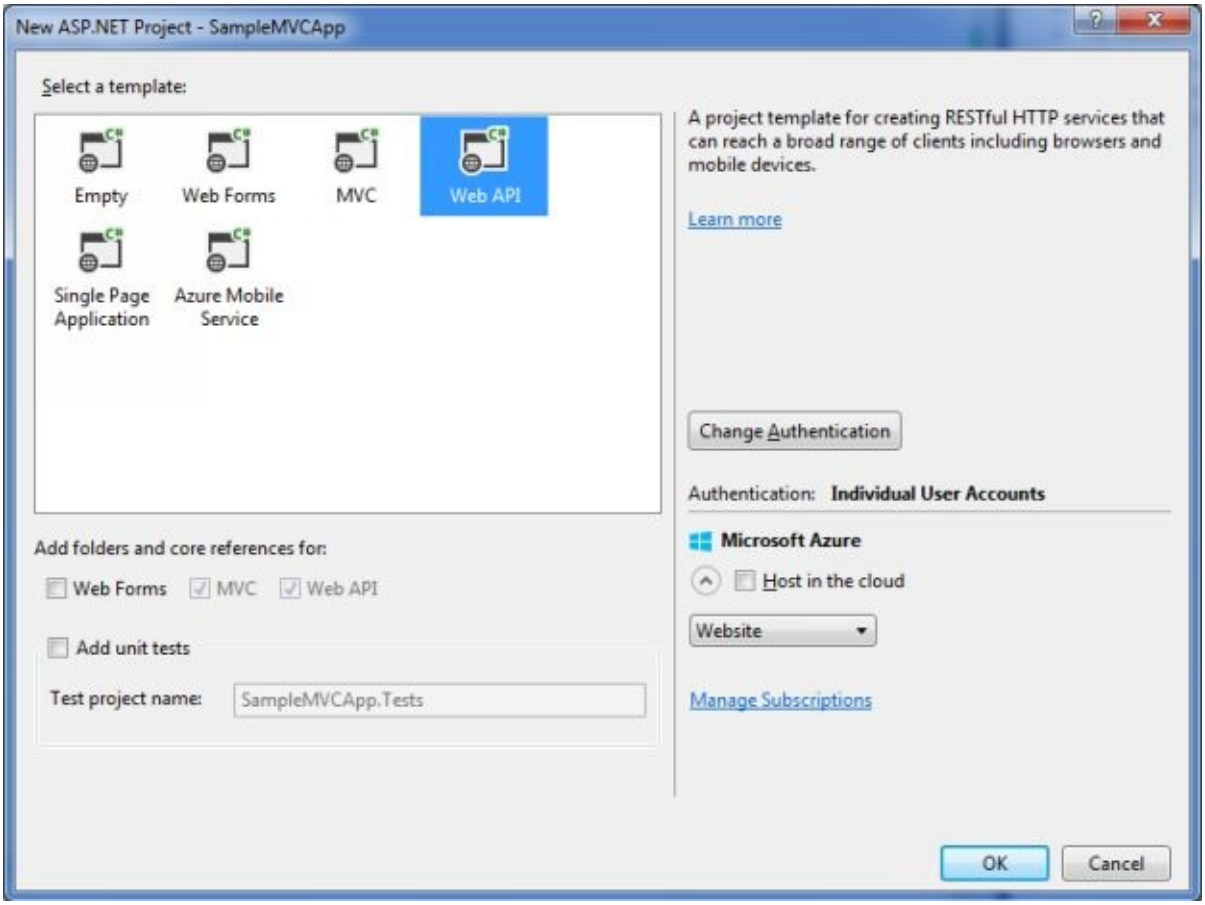

Figure 1: Creating an ASP.NET MVC project based on the Web API template

Next, we'll create a sample model inside the model folder in the MVC solution. Right click on the Model folder from the solution explorer and select Add -> Class as shown in Figure 2.

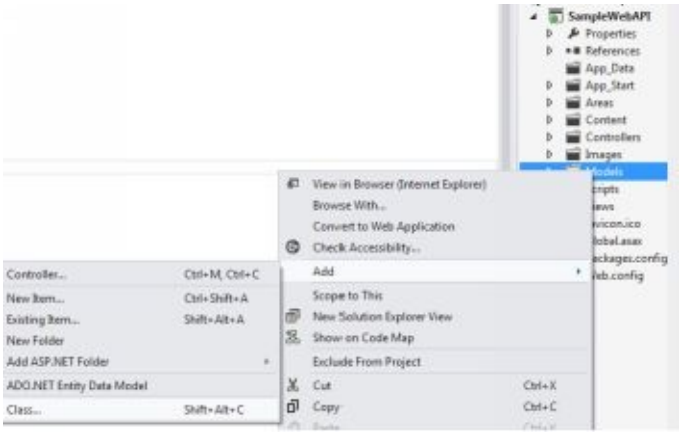

Figure 2: Add new Model

For this walk-through, I am using the **Product** model defined in Listing 1.

Listing 1: Defining the Product model

```
public class Product
    {
    public int Id { get; set; }
    public string Name { get; set; }
    public string Category { get; set; }
    public decimal Price { get; set; }
    }
```

Once the product model is ready, let us create a new API controller inside the controllers' folder to process the **Product** model. By default, an MVC project based on the Web API template adds two controllers: one inherited from the **Controller** class and the other from **ApiController** class.

Right-click the controllers folder in Solution Explorer and add a new controller by selecting the **Empty API controller** option under template as shown in Figure 3.

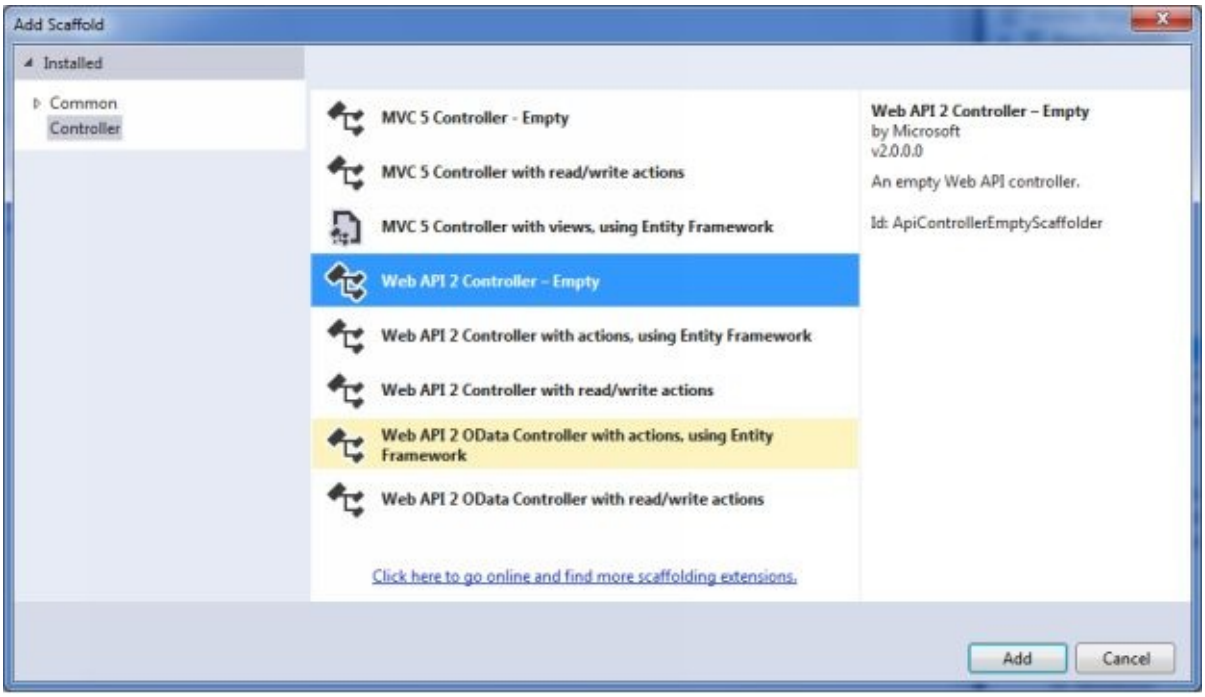

Figure 3: Add Empty API Controller

Provide appropriate name for the new controller; say ProductsController.

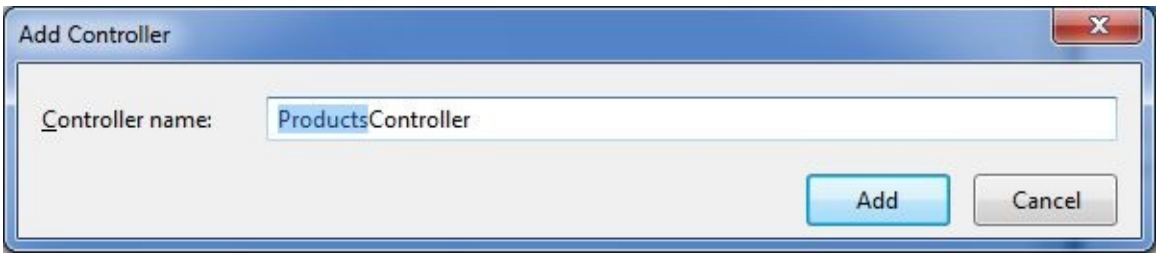

Figure 4: Add Controller

ProductsController will define two methods for getting list of products and selected

product based on product id. The code for the sample controller should look like that shown in Listing 2.

Listing 2: Adding an ApiController class to the sample project

```
public class ProductsController : ApiController
    {
    //Define the products list
    List<Product> products = new List<Product>();

    /// <summary>
    ///Web API method to return list of products
    /// </summary>
    /// <returns></returns>
    public IEnumerable<Product> GetAllProducts()
    {
    GetProducts();
    return products;
    }

    private void GetProducts()
    {
    products.Add(new Product { Id = 1, Name = "Television", Category = "Electronic", Price = 82000 });
    products.Add(new Product { Id = 2, Name = "Refrigerator", Category = "Electronic", Price = 23000 });
    products.Add(new Product { Id = 3, Name = "Mobiles", Category = "Electronic", Price = 20000 });
    products.Add(new Product { Id = 4, Name = "Laptops", Category = "Electronic", Price = 45000 });
    products.Add(new Product { Id = 5, Name = "iPads", Category = "Electronic", Price = 67000 });
    products.Add(new Product { Id = 6, Name = "Toys", Category = "Gift Items", Price = 15000 });
    }

    /// <summary>
    /// Web API method to retrurn selected product based on the passed id
    /// </summary>
    /// <param name="selectedId"></param>
    /// <returns></returns>
    public IEnumerable<Product> GetProducts(int selectedId)
    {
    if (products.Count() > 0)
    {
```

```
return products.Where(p => p.Id == selectedId);
}
else
{
GetProducts();
return products.Where(p => p.Id == selectedId);
}
}
}
```

Run the project and access the API by appending the uniform resource locator (URL) with **/api/Products,** as in **http://localhost: 59509/api/Products.** This api call will return the list of Products. We can also access the **GetProducts** method by passing the **SelectedId** argument as part of the query string: **http://localhost:59509/api/Products?SelectedId=2**.

By default Web API returns JSON data, so when you run the Web API directly from browser, it will prompt to save or open the JSON output. Following Figure shows the api output for the /api/Products request.

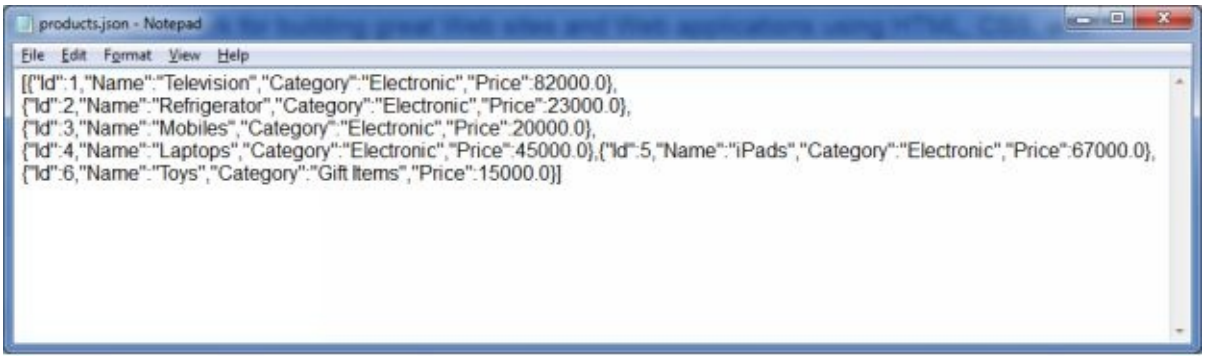

Figure 5: Web API Response - JSON format

Passing Complex Objects to a Web API Method

Passing a simple value is a straightforward process in Web API, but in most cases you'll need to pass a complex object as a method parameter. You can pass these objects either by using the **[FromUrl]** or by using **[FromBody]** attribute. The **[FromBody]** attribute reads data from the request body. However, the attribute can be used only once in the method parameter list.

Let us understand the importance of using complex parameters using the code snippet in Listing 3, which expects a complex object, **Time**, as its input.

Listing 3: Passing a complex object as a method parameter

```
public class SampleController : ApiController
    {
    /// <summary>
    /// Get the time based on passed parameters
    /// </summary>
    /// <param name="t"></param>
    /// <returns></returns>
    public string GetTime(Time t)
    {
    return string.Format("Received Time: {0}:{1}.{2}", t.Hour, t.Minute, t.Second);
    }
    }

    public class Time
    {
    public int Hour { get; set; }
    public int Minute { get; set; }
    public int Second { get; set; }
    }
```

Now, let us try to pass the values to the **GetTime** method using the **Hour**, **Minute** and **Second** values. This will throw the Object reference not set to an instance of an object exception.

Even though we have mentioned the values related to the fields of the Time object, API method is not able to map the values properly to the parameter. Now modify the code to include the **[FromUri]** attribute so it can handle the complex object from query string. As you can see in the following snippet, we include the **[FromUri]** attribute before specifying the **Time** object:

```
public string GetTime([FromUri] Time t)
```

{ ———-}

Invoke the web API method with the same values before and observe the result, as shown in Figure 5.

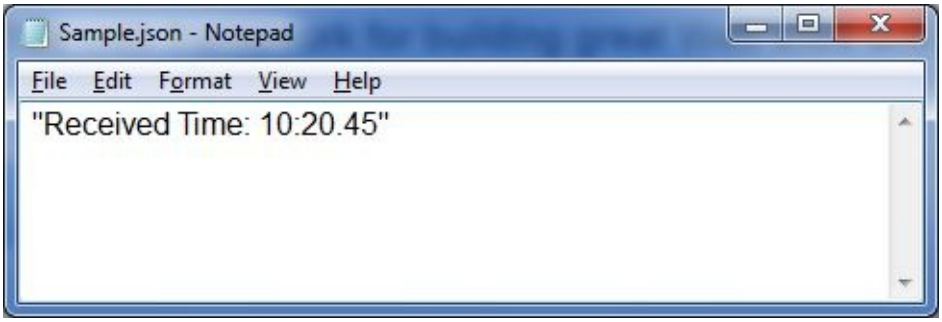

Figure 6: Receiving successful results when calling the GetTime method

Web API Client Implementations

Web API can be consumed from different applications including web applications, client side technologies like jQuery and AngularJS, native mobile applications and so on. This section discuss about how we can invoke Web API from various client applications; ASP.NET MVC and jQuery.

Working with the HttpClient API

HttpClient is an extensible API for accessing any services or web sites exposed over HTTP. The HttpClient API was introduced as part of the WCF Web API but is now available as part of ASP.NET Web API in.NET Framework 4.5. You can use HttpClient to access Web API methods from a code-behind file and from services such as WCF.

The code snippet shown in Listing 4 creates an HttpClient object and uses it for asynchronous access to products API methods. Refer the code comments to understand the code snippet.

Listing 4: Creating an HttpClient object to access sample API methods

```
// asynchronous accessing of web api method
async Task GetData()
{
StringBuilder result = new StringBuilder();
// Define the httpClient object
using (HttpClient client = new HttpClient())
{
// Define the base address of the MVC application hosting the web api
// accessing the web api from the same solution
client.BaseAddress = new Uri(HttpContext.Current.Request.Url.AbsoluteUri);
// Define the serialization used json/xml/ etc
client.DefaultRequestHeaders.Accept.Add(
new MediaTypeWithQualityHeaderValue("application/json"));
// Call the method
HttpResponseMessage response = client.GetAsync("api/products").Result;
if (response.IsSuccessStatusCode)
{
// Convert the result into business object
var products = response.Content.ReadAsAsync<IEnumerable<Product>>().Result;
foreach (var p in products)
{
result.Append(string.Format("{0} – [Price: {1} Category: {2}] ", p.Name, p.Price, p.Category));
}
}
else
{
result.Append(string.Format("Error                                                    Code:-
{0}  Error Details: {1}", (int)response.StatusCode, response.ReasonPhrase));
```

```
}
}

data.Text = result.ToString();
}
```

Accessing the Web API from jQuery

From an HTML5 Web application, you can use jQuery to directly access the Web API. You call the API by using the **getJSON()** method to extract the business object from the result. jQuery can identify internal fields associated with the business object, such as **data[i].Price.**

The code snippet in Listing 5 retrieves data from our **Product** API methods and displays the data in a tabular format. Refer the code comments to understand the code snippet.

Listing 5: Using jQuery to retrieve data from the Web API methods

```
<head>
    <title></title>

    <script src="Scripts/jquery-1.8.2.js"></script>
    <script type="text/javascript">
    $.getJSON("api/products",
    function (data) {
                //Clear the div displaying the result
    $("#productView").empty();
                    //Create a table and append the table body
    var $table = $('<table border="2">');
    var $tbody = $table.append('<tbody />').children('tbody');

                    //data - return value from the Web API method

    for (var i = 0; i < data.length; i++) {

                        //add a new row to the table
    var $trow=$tbody.append('<tr />').children('tr:last');
                        //add a new column to display Name
    var $tcol = $trow.append("<td/>").children('td:last')
    .append(data[i].Name);
                        //add a new column to display Category
    var $tcol = $trow.append("<td/>").children('td:last')
    .append(data[i].Category);
                        //add a new column to display Price
    var $tcol = $trow.append("<td/>").children('td:last')
    .append(data[i].Price);
    }
```

```
                    //display the table in the div
    $table.appendTo('#productView');
    });
    </script>
</head>
<body>
    <div id="productView"></div>
</body>
</html>
```

Passing a Parameter from jQuery

In addition to retrieving data from the Web API methods, you can use jQuery to pass parameters back to the Web API. jQuery supports several methods for passing the parameters. The first approach is to use separate parameter set. The following code snippet demonstrates how to pass a single parameter:

```
$.getJSON("api/products",
{ selectedId: '4' },
function (data) {    ….});
```

You can also pass multiple parameters within a parameter set, as shown in the following snippet:

```
$.getJSON("api/products",
{ selectedId: '4', name: 'TV' },
function (data) {    ….});
```

Another approach you can take is to pass a parameter by appending it to the URL query string, as shown in the following snippet:

```
$.getJSON("api/products?selectedId =4",
    function (data)  {    ….});
```

First approach separates the query string or parameter passing from the URL and is appropriate for complex or multiple data passing. Second approach is more suitable for passing one or two parameters.

Scaffolding with Web API

ASP.NET Scaffolding is a code generation framework for ASP.NET Web applications. Visual Studio includes pre-installed code generators for MVC and Web API projects. Visual Studio automatically generates the API code required to perform various operations on the specified model or data source.

Scaffold on the Entity Framework

To demonstrate how scaffolding works in a Web API project, we can use the Entity Framework to define our database and generate the Web API methods. First, we need to add an ADO.NET entity data model that defines the data.

Add an ADO.NET Entity Data Model to the project using the Add new Item context menu.

Figure 7: New ADO.NET Entity Data Model

Select the Empty EF designer model option to define the models using Model First approach.

NOTE: For more details on different entity framework approaches, refer Different Approaches of Entity Framework

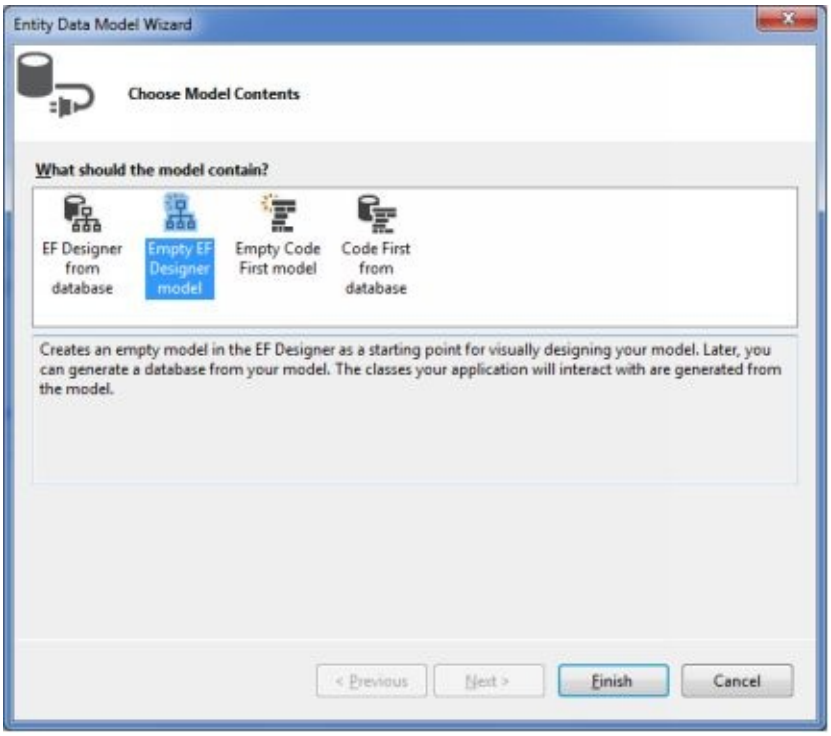

Figure 8: Empty EF Designer model

Design the models using the Entity Data Model designer. For the sample, use the data

model based on two tables, **Course** and **Status**, as shown in the next Figure.

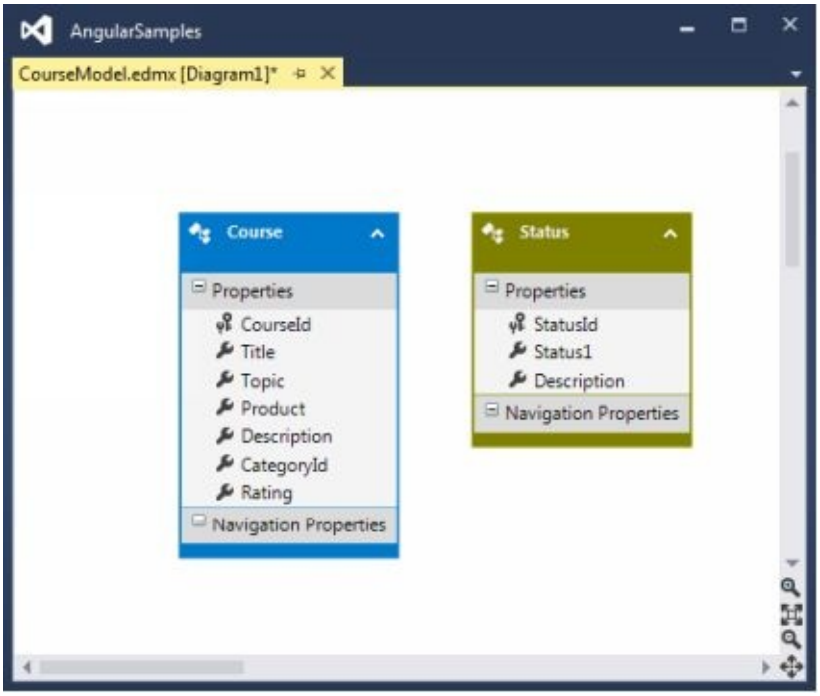

Figure 9: Adding an entity data model based on the Course and Status tables

After we define the data model, we can generate the Web API controller that corresponds to the model. Rebuild the application to generate the code context corresponding to the EF data model. Right-click **Controller** and then click **Add Controller**, which lunches the **Add Scaffold** dialog box. Select the **Web API 2 Controller with actions, using Entity Framework** option and click Add.

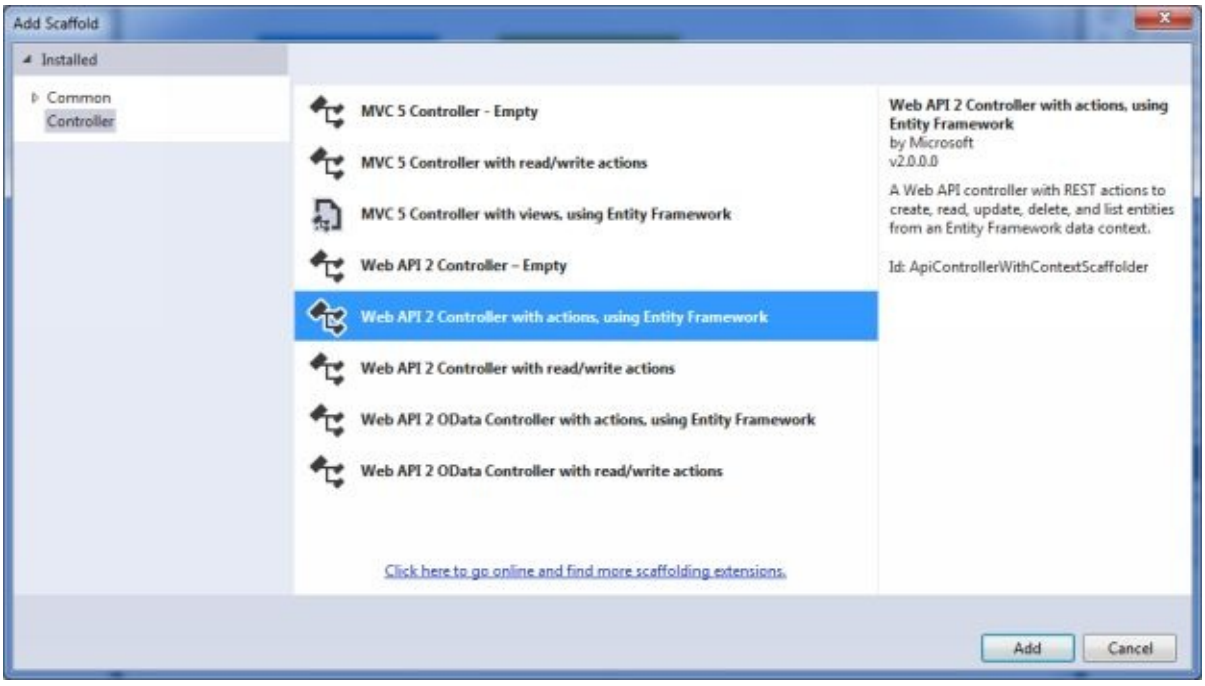

Figure 10: Adding a controller based on the EF data model

This will open the **Add Controller** window. Select the Course model from the list of available models and select the container corresponding to the course model for Data Context.

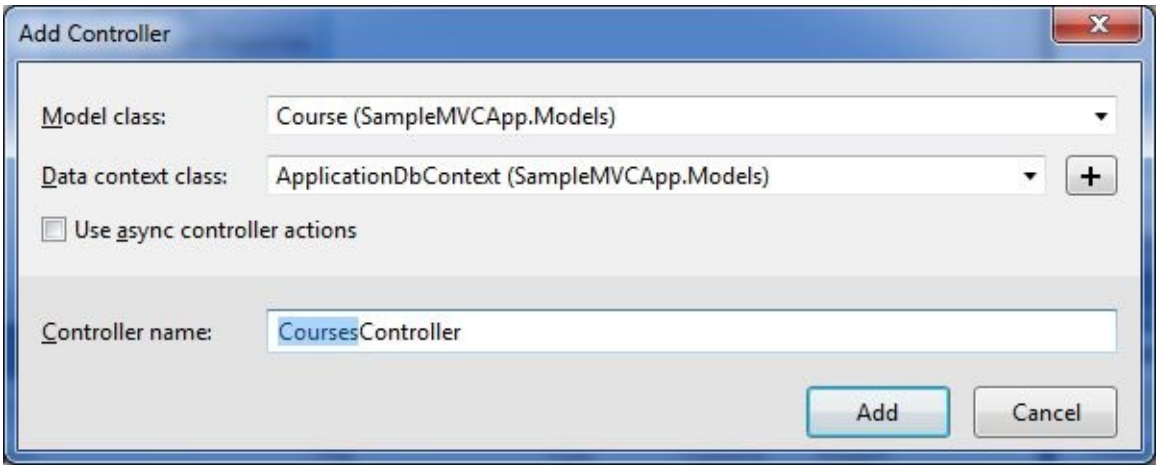

Figure 11: Add Controller

If you are planning to execute the Web API methods in asynchronous manner, select the **Use async controller actions** checkbox. Click on Add to add the new Web API controller, courseController based on actions related to the Course model. Same way add the Web API methods for the Status model too.

Listing 6 shows the automatic code generated for the **CourseController** API controller. Notice that it contains methods to support get, put, post, and delete operations that correspond to the selected model.

Listing 6: Code generated for the CourseController API controller

```
public class CoursesController : ApiController
    {
    private ApplicationDbContext db = new ApplicationDbContext();

    // GET: api/Courses
    public IQueryable<Course> GetCourses()
    {
    return db.Courses;
    }

    // GET: api/Courses/5
    [ResponseType(typeof(Course))]
    public IHttpActionResult GetCourse(int id)
    {
    Course course = db.Courses.Find(id);
    if (course == null)
    {
    return NotFound();
    }
```

```
return Ok(course);
}

// PUT: api/Courses/5
[ResponseType(typeof(void))]
public IHttpActionResult PutCourse(int id, Course course)
{
if (!ModelState.IsValid)
{
return BadRequest(ModelState);
}

if (id != course.CourseId)
{
return BadRequest();
}

db.Entry(course).State = System.Data.Entity.EntityState.Modified;

try
{
db.SaveChanges();
}
catch (DbUpdateConcurrencyException)
{
if (!CourseExists(id))
{
return NotFound();
}
else
{
throw;
}
}

return StatusCode(HttpStatusCode.NoContent);
}
```

```
// POST: api/Courses
[ResponseType(typeof(Course))]
public IHttpActionResult PostCourse(Course course)
{
if (!ModelState.IsValid)
{
return BadRequest(ModelState);
}

db.Courses.Add(course);
db.SaveChanges();

return CreatedAtRoute("DefaultApi", new { id = course.CourseId }, course);
}

// DELETE: api/Courses/5
[ResponseType(typeof(Course))]
public IHttpActionResult DeleteCourse(int id)
{
Course course = db.Courses.Find(id);
if (course == null)
{
return NotFound();
}

db.Courses.Remove(course);
db.SaveChanges();

return Ok(course);
}

protected override void Dispose(bool disposing)
{
if (disposing)
{
db.Dispose();
}
```

```
base.Dispose(disposing);
}

private bool CourseExists(int id)
{
return db.Courses.Count(e => e.CourseId == id) > 0;
}
}
```

The Web API scaffolding feature reduces the development effort required to generate the API methods necessary to perform CRUD (create, read, update, delete) operations in different models.

Routing in ASP.NET Web API

Web API uses routing to match uniform resource identifiers (URIs) to various actions. The **WebApiConfig** file, located inside the **App_Start** node in Solution Explorer, defines the default routing mechanism used by Web API. The mechanism is based on a combination of HTTP method, action, and attribute. However, we can define our own routing mechanism to support meaningful URIs.

For example, in our sample Web API project, we use **/api/Products** to retrieve product details. However, we can instead use **api/Products/GetAllProducts** by modifying the default routing logic to include **{action}** as part of the URI, as shown in the following code snippet:

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

After we update the route, we can use the **ActionName** attribute in our Web API method to specify the action name, as the follow snippet shows:

```
[ActionName(“GetAllProducts”)]
public IEnumerable<Product> GetAllProducts()
{……..}
```

NOTE: Use the **[NonAction]** attribute to indicate that the Web API method is not an API action method.

We can also use an HTTP method or **Acceptverbs** attribute to specify the HTTP action required to access the Web API method. For example, the following snippet uses the **HttpGet** and **HttpPost** methods:

```
[HttpGet]
public IEnumerable<Product> GetProducts(int selectedId)
{…….. }

[HttpPost]
public void AddProduct(Product p)
{…….. }
```

For multiple methods, use the **Acceptsverbs** attribute, as shown in the following snippet:

```
[AcceptVerbs(“GET”,“PUT”)]
public IEnumerable<Product> GetProducts(int selectedId)
```

{…….. }

Attribute Routing

Web API 2 introduced the attribute routing. Earlier versions support the conventional routing mechanism discussed in last section. This approach is good for maintaining the routing logic in one place and apply consistently across the application. But, this approach lack the flexibility of defining the routing for objects with sub groups like Products may have different categories, books will have authors, and so on. Attribute routing address this issue and provides the flexibility to define the routing in granular levels.

Enable the routing

For enabling the attribute routing in Web API, call the MapHttpAttributeRoutes during the configuration. Open the **WebApiConfig** file, located inside the **App_Start** node in Solution Explorer. Following entry define the attribute routing capability.

```
        // Web API routes
    config.MapHttpAttributeRoutes();
```

This entry will be available in ASP.NET MVC 5 based project; if not available add the above statement to enable the attribute routing.

Add Route

Add the route attribute to the Web API method to define attribute routing for specific Web API methods

```
[Route("ProductDetail/{selectedId}")]
public IEnumerable<Product> GetProducts(int selectedId)
    {……..}
```

Above method can be accessed now using **http://localhost:59509/ProductDetail/2.** Notice that the new URL not even have the api or controller names.

Route Prefix

Most of the time the controller will have the same format for the routes for all the methods.

```
public class ProductsController : ApiController
    {
    [Route("api/products")]
    public IEnumerable<Product> GetProducts() { … }


    [Route("api/products/{id}")]
    public Product GetProduct(int id) { … }
    }
```

We can use a RoutePrefix to avoid the duplicate entry of prefixes for the web API method Route attributes.

```
[RoutePrefix("api/products")]
```

```
public class ProductsController : ApiController
{
[Route("")]
public IEnumerable<Product> GetProducts() { … }

[Route("{id}")]
public Product GetProduct(int id) { … }
}
```

RoutePrefix will be applied in controller level.

Route Constraint

Route constraints help us to restrict the parameter processing. Consider the about attribute routing where we have defined a parameter 'id', without any constraint. Even if the client send a string value, the routing will try to match the parameter and throw an exception due to type mismatch. To avoid such runtime exceptions, we can define a constraint for the route parameters.

Route constraint will be defined as

{parameter: constraint}

Eg:-

Integer constraint - {id:int}

Integer with maximum value as 25 – {id:max(25)}

Datetime constraint – {startDate: datetime}

Guid constraint – {id:guid}

We can apply multiple constraints to a parameter by separating each constraint using a colon.

{id:int:min(1)} – indicate an integer with a minimum value of 1.

Implementing Multiple Serialization Options

Web API supports numerous serialization methods, including XML and JSON. In the sections to follow, we'll take a closer look at how to implement these three methods and then compare their behavior. Before doing that, however, let's modify the sample Web API project to return bulk data so we can better understand and compare the performance of the serialization methods. Listing 7 shows how the GetProducts() method updated to return bulk data.

```
private void GetProducts()
{
for (int i = 0; i < 5000; i++)
{
products.Add(new Product { Id = i, Name = "Product -  "+i,
Category = "The ASP.NET and Visual Web Developer teams have released the ASP.NET and Web Tools 2012.2 upda
Price = 1 });
}
}
```

Listing 7: Updating the GetProducts() method

Using JSON Serialization

Web API uses JSON as its default serialization method. As a result, when you run the Web API project, the results are automatically returned in the JSON format. For example, to retrieve details from **Products**, you need only use the syntax **http://<<server>>:<<port>>/api/Products** to return the results in the JSON format.

Using XML Serialization

To use the XML serialization method in a Web API project, you must modify the **Global.asax** file by inserting the following two lines of code at the end of **Application_Start()** method:

```
GlobalConfiguration.Configuration.Formatters.RemoveAt(0);
GlobalConfiguration.Configuration.Formatters.XmlFormatter.UseXmlSerializer = true;
```

After implementing the XML serialization method, you can retrieve details from **Products** by using the syntax **http://<<server>>:<<port>>/api/Products**, the same syntax used for JSON.

Help Page Generation from XML Documentation

By default, Web API provides help pages related to the API methods defined in the project. We can generate the help pages from the XML documentations provided as part of the method definition.

To create the help pages, expand the **Areas** node in Solution Explorer and then open the **HelpPageConfig.cs** file

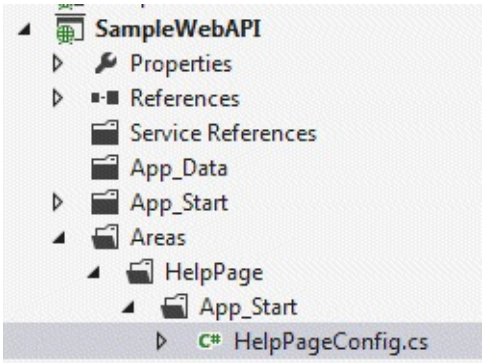

Figure 12: Accessing the HelpPageConfig.cs file in the sample Web API project

In the **HelpPageConfig.cs** file, un-comment the following statement:

```
config.SetDocumentationProvider(new XmlDocumentationProvider(HttpContext.Current.Server.MapPath("~/App_Data/
```

Next, enable XML documentation by selecting the **XML documentation file** option in the **Build** section of the project's properties, as shown in Figure 11. Copy the filename specified in the config file to the XML documentation file option.

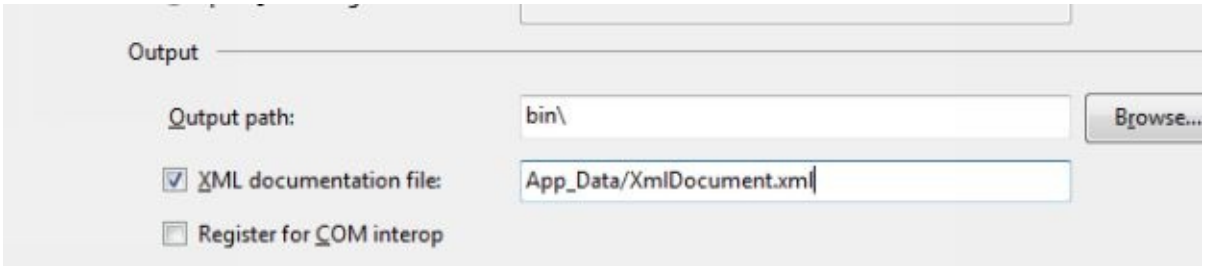

Figure 13: Enabling the XML documentation file in the Web API project

Now, let's add the XML documentation to our sample API controller, as shown in Listing 9. Just before each method definition, specify "///" to get the default XML documentation for the method. Modify further with more specific comments.

```
/// <summary>
/// Get All products for our Sample
/// </summary>
/// <returns></returns>
public IEnumerable<Product> GetAllProducts()
{
        ____
}
```

```
/// <summary>
/// Return the details of selected product depends on the product id passed.
/// </summary>
/// <param name="selectedId"></param>
/// <param name="name"></param>
/// <returns></returns>
public IEnumerable<Product> GetProducts(int selectedId)
{
        ______

}
```

Listing 9: Adding XML documentation to the sample API controller

Once you've updated the controller, run the application and navigate to **API** link at the top-right corner of the browser screen, as shown in next Figure.

Figure 14: Viewing the API link in the application window

Click the **API** link to open the help page. Notice the XML documentation comments that had been added to the controller, as shown in next Figure.

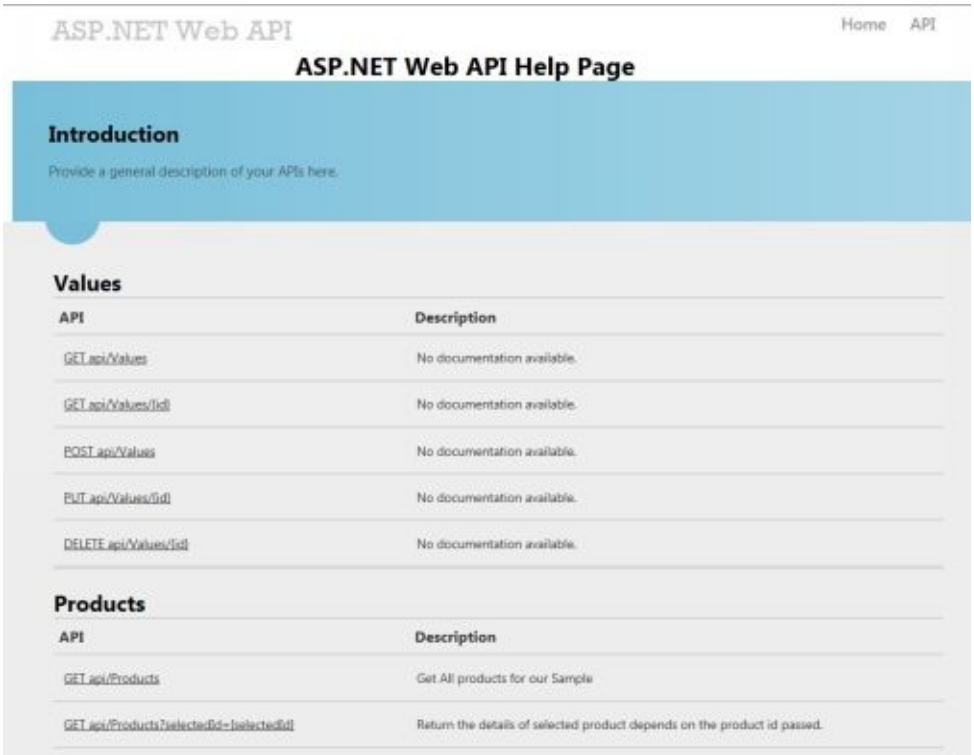

Figure 15: Viewing the comments in the Web API help file

WCF via Web API

WCF supports numerous protocols and messaging formats. With Web API, we can create only HTTP-based services. Obviously, we can create HTTP services with WCF, but they're hidden inside different abstraction layers. In addition, implementation is more complex with WCF than with Web API. HTTP implemented on WCF uses the HTTP post model, and we may not be able to use the main features associated with HTTP protocols such as Caching and Status Code. By using Web API, we can perform all activities like get, post, and put, and we can use different HTTP protocol features.

WCF requires complex binding and address manipulation and configurations, but Web API is straightforward and simple to use. Web API is also a better choice for RESTfull service creation. You can find a detailed comparison between WCF and Web API in the MSDN article "WCF and ASP.NET Web API."

References

1. [Different Approaches of Entity Framework]

   https://www.simple-talk.com/dotnet/.net-framework/different-approaches-of-entity-framework/

2. [WCF and ASP.NET Web API]

   http://msdn.microsoft.com/en-us/library/jj823172.aspx