Azure Fast Start for Mobile Application Development

Module 6: Azure API Apps

Student Lab Manual

Instructor Edition (Book Title Hidden Style)

Version 1.0

**Conditions and Terms of Use**

Microsoft Confidential - For Internal Use Only

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

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

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

© 2015 Microsoft Corporation. All rights reserved.

**Copyright and Trademarks**

© 2015 Microsoft Corporation. All rights reserved.

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

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

For more information, see Use of Microsoft Copyrighted Content at  
<http://www.microsoft.com/en-us/legal/intellectualproperty/Permissions/default.aspx>

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

Contents

[Lab 1: Creating a New Microsoft Azure App 1](#_Toc432976118)

[Exercise 1: Creating the solution 3](#_Toc432976119)

[Task 1: Creating the New Azure API App 3](#_Toc432976120)

[Task 2: Adding the Code to Represent the Product Catalogue 5](#_Toc432976121)

[Task 3: Enable Swagger UI 7](#_Toc432976122)

[Task 4: Setting Up CRUD Operations 10](#_Toc432976123)

[Task 5: Advanced Data Queries for Products 13](#_Toc432976124)

[Exercise 2: Deploying the solution 16](#_Toc432976125)

[Task 1: Configure Application Metadata 16](#_Toc432976126)

[Task 2: Create the Application in Azure 17](#_Toc432976127)

[Task 3: Deploy the Application's Code to Azure 20](#_Toc432976128)

[Task 4: Access the Newly Created Azure API App 22](#_Toc432976129)

[Lab 2: Migrating an Existing WebAPI App 24](#_Toc432976130)

[Exercise 1: Creating the WebAPI application 25](#_Toc432976131)

[Task 1: Creating the New WebAPI App 25](#_Toc432976132)

[Task 2: Adding the Code to Connect to SQL Server Azure and Retrieve Products 26](#_Toc432976133)

[Task 3: Adding the Image Upload / Download-Related Helper Classes 29](#_Toc432976134)

[Exercise 2: Converting the application to Azure API 39](#_Toc432976135)

[Task 1: Adding the Nuget Packages Needed for Conversion to Azure API 39](#_Toc432976136)

# Lab 1: Creating a New Microsoft Azure App

#### Introduction

In this lab, you will learn how to create a new Azure application programming interface (API) App. You will create the application from scratch using Microsoft Visual Studio 2015, run it locally, learn about its configuration features, and add code to create APIs for accessing the product catalogue located in SQL Server Azure. Finally, you will use Swagger to render the API more discoverable and processed to deploy the newly built app to the Azure cloud as an Azure API App*.*

#### Objectives

After completing this lab, you will be able to:

* Know which templates to use to create an Azure API App using Visual Studio 2015.
* Explain how Azure API Apps use Swagger to render API discoverable.
* Know how to declare the API App in the Azure portal.
* Deploy the API App to the Azure cloud.

#### Prerequisite

Lab M03 – Azure SQL Database : The **ProductsDB** database will be reused in this lab.

#### Estimated time to complete this lab

45 minutes

#### Scenario

For this example, the aim is to build an Azure API App that will expose a product catalogue of an online store. Many real-world online stores expose a product catalogue online through API, so that price comparison websites and shopping search engines can then use the API and index the catalogue.

The catalogue will contain the listings of categories / sub-categories / products, and products pictures through Representational State Transfer (REST) API in the cloud. Other websites and applications will then be able to consume these services to gain information about the products we intend to make available online and their categorization.

To achieve this, we will:

* Start by building an Azure API App from scratch using Visual Studio.
* Add the classes that will represent the entities in our online catalogue: categories, products, pictures, etc.
* Code the controllers that will expose information about the catalogue to the consuming clients (other websites or applications).
* Connect the application to an existing SQL Server Azure database that already contains the product catalogue tables and data.
* Test that we are able to retrieve data using the newly built application.
* Deploy the newly built application to the Azure cloud as an Azure API App.

## Exercise 1: Creating the solution

#### Objectives

In this exercise, we will create a new Azure API App using one of the built-in templates that comes with Visual Studio 2015 and then run the application locally to make sure that you are familiar with its components and the way the application is supposed to work. We will then proceed to add the connection to the SQL Server Azure database and the business logic layer code to expose the data in this database (the product catalogue) through the services.

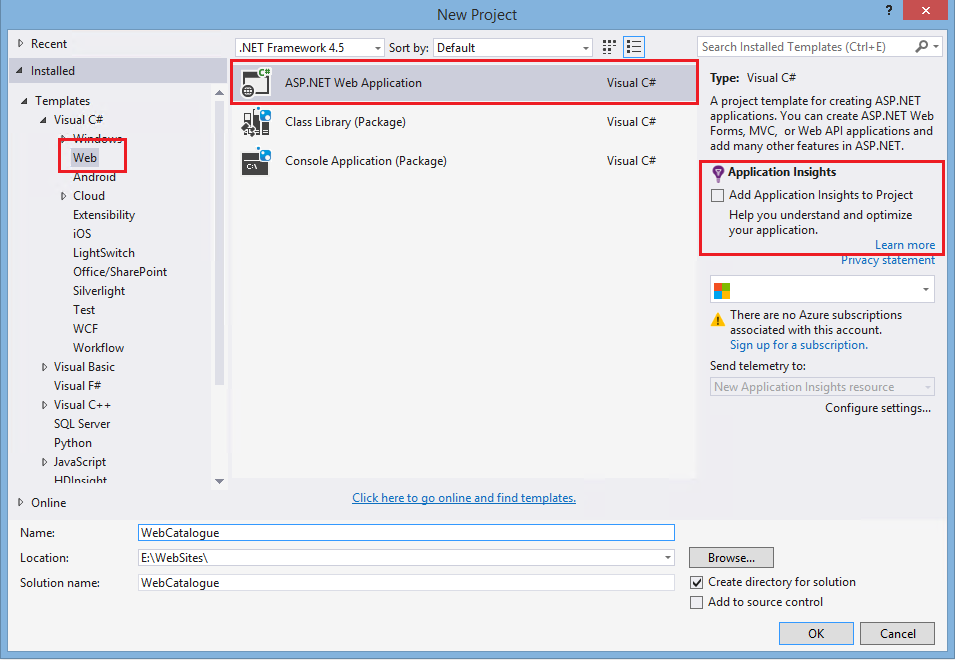
#### Scenario

In order to create the online product listings for our product catalogue, we will start by instantiating a new Azure API App using Visual Studio to have a stub on which we can build and expose our business logic on. Once the stub is created, we will:

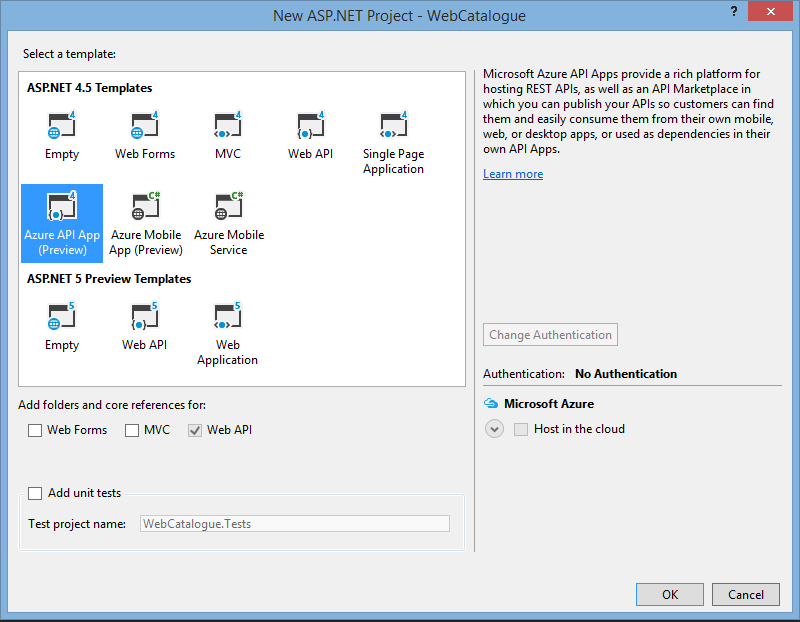
* Connect the application to SQL Server Azure.
* Create the entities to represent the data in the SQL Server Azure database catalogue.
* Code the controllers to expose the data on the Internet.
* Use the Swagger UI to check if the services work as expected.

### Task 1: Creating the New Azure API App

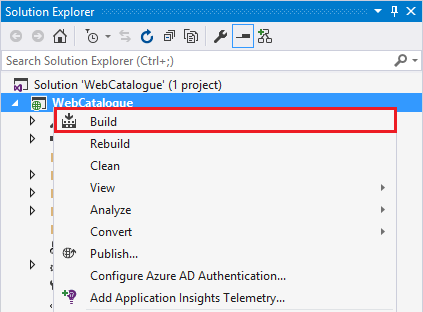
1. Open Visual Studio 2015, and select **File** > **New** > **Project**.
2. In the **New Project** dialog box, in the left pane, expand **Templates** > **Visual C#** and then select **Web**.



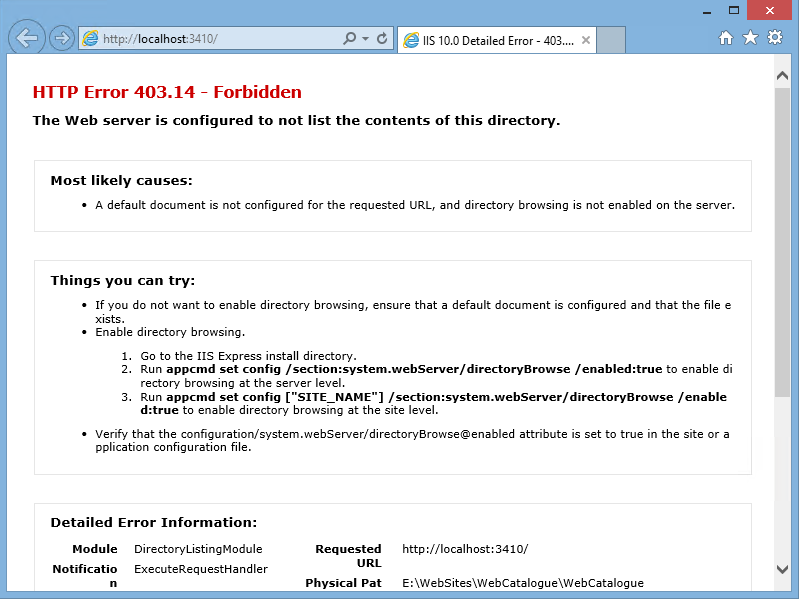
1. In the **New Project** dialog box, in the middle pane, select **ASP.NET Web Application**. Make sure you clear the **Add Application Insights to Project Help you understand and optimize your application** check box..
2. In the **New Project** dialog box, in the **Name** box, enter **WebCatalogue**, and in the **Location** box, chose a folder on your local machine where the solution files are to be created and stored by Visual Studio, and click **OK**.
3. In the **New ASP.NET Project – WebCatalogue** dialog box, under **ASP.NET 4.5 Templates**, click **Azure API App (Preview)**.



1. In the **New ASP.NET Project – WebCatalogue** dialog box, click **OK** for Visual Studio to start generating the solution and its corresponding files.
2. Once the solution is generated by Visual Studio, go to the Solution Explorer, click the **WebCatalogue** project, and click **Build** to have the project compile for the first time.



1. Next, press Ctrl+F5 to run the application for the first time. An Internet Explorer (or Edge) window will be displayed with the message **HTTP Error 403.14 – Forbidden**. This is normal and expected.



### Task 2: Adding the Code to Represent the Product Catalogue

1. Open the WebCatalogue solution and in the Solution Explorer, double-click the web.config file. In the web.config file, find the <connectionStrings> tag. If the section does not exist, create the section right after the end of the <appSettings> section as such:

<connectionStrings>

</connectionStrings>

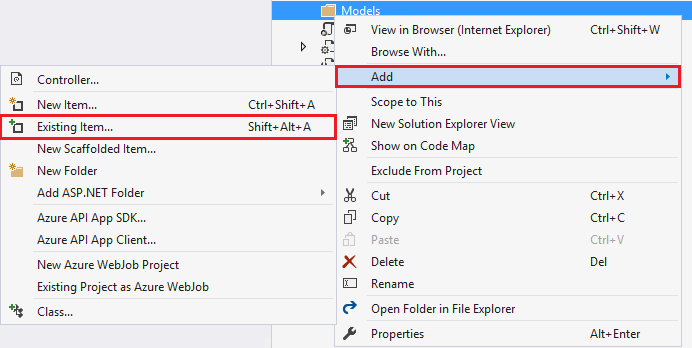
1. Add the following connection string to be able to connect to the ProductsDB database in SQL Server Azure.

<connectionStrings>

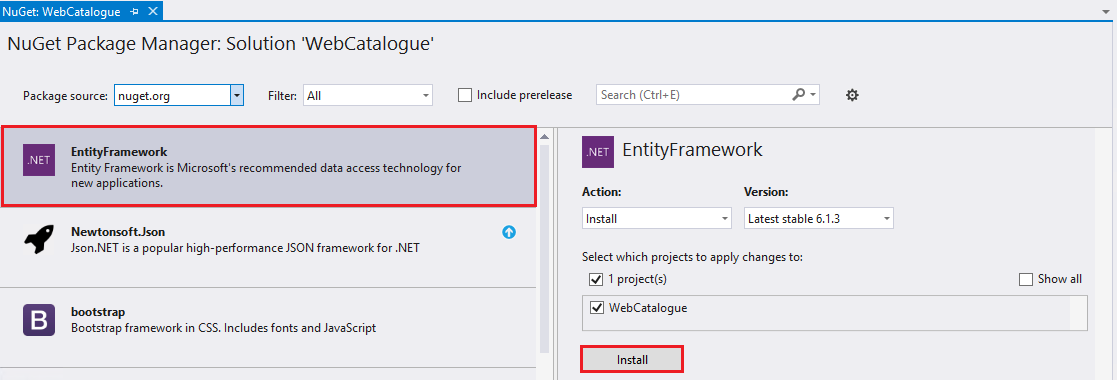
<add name="DBConnection" providerName="System.Data.SqlClient" connectionString="Server=tcp:{your\_server\_name}.database.windows.net,1433;Database=ProductsDB;User ID=ProductsDBAdmin@{your\_server\_name};Password={your\_password\_here};Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;" />

</connectionStrings>

1. In the Solution Explorer in Visual Studio, select the **Models** folder. Right-click the folder and from the shortcut menu, select **Add** > **Existing Item**. From the **begin** folder corresponding to this exercise, select the **DataLayer.cs** **and DataModel.cs** files.



1. Inspect the **DataModel.cs** class file. You will see that it declares the following classes: ProductCategory, ProductSubCategory, ProductSpecification, and Product. The way that the information is organized in the database means that the web-catalogue will have categories of products. Each category will have one or more sub-categories. The sub-categories have products and the products have product specifications.
2. Inspect the **DataLayer.cs** class file. You will see that there are some pieces of code that are underlined in red. This is because we are missing Entity Framework, the ORM that will allow us to connect our entities to the SQL Server Azure database.
3. Click **Tools** > **Nuget Package Manager** > **Manage NugetPackages for Solution**, select **EntityFramework** from the list, and click **Install**.

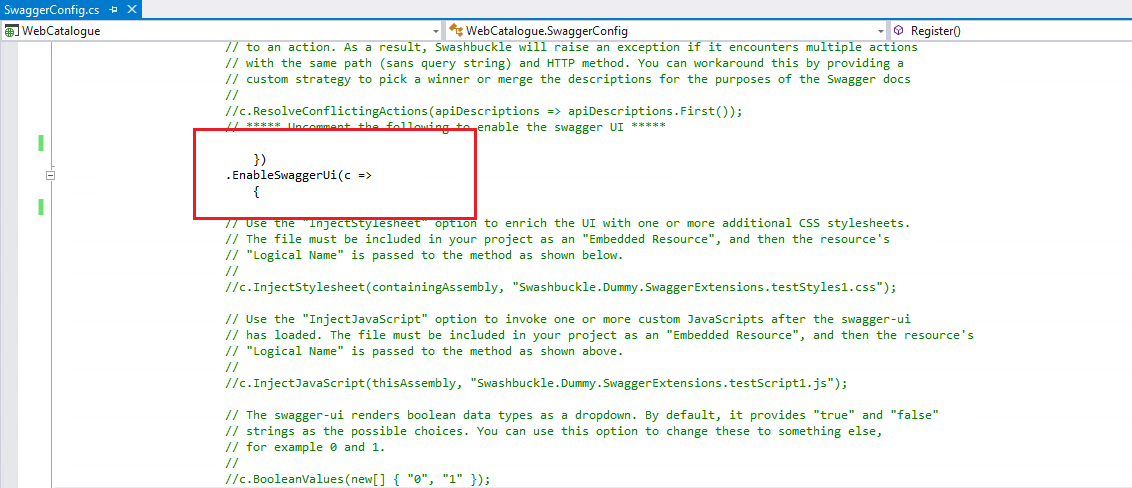


1. In the Solution Explorer in Visual Studio, select the **Controllers** folder, right-click the folder, and from the shortcut menu, select **Add** > **Existing Item**.
2. From the **Start** folder of this lab, select the following files: ProductCategoriesController.cs, ProductsController.cs, ProductsCsvController.cs, ProductSpecificationsController.cs, and ProductSubCategoriesController.cs and add them to the project.
3. Inpsect the **ProductsController.cs** class. This is a WebAPI controller. It declares a private variable of type EFContext, which is the Entity Framework data context that will allow it to extract data from the Azure SQL Server Database. It then implements action methods like the GetProducts() which will use the context to return the list of all products available in the database, serialize them to JavaScript Object Notation (JSON), and send them to the caller.
4. Launch the application again, and this time, add the following trailing segments to the URL that is open in the browser: /api/Products. This will cause a GET request to be sent to the application, which will be routed to the ProductsController class and will trigger the GetProducts() action method. You should receive a JSON file, which you can inspect in Visual Studio to see the list of products.

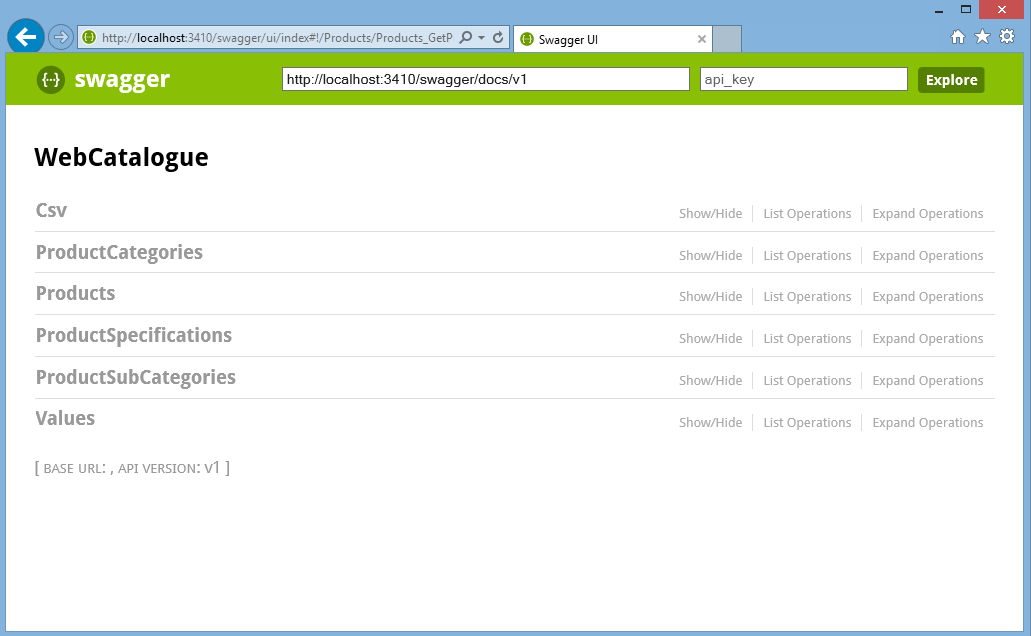
### Task 3: Enable Swagger UI

API services are available in all forms and shapes. Many API services do not have a standardized way of returning data or discovering various REST URLs they expose to the client. This is where Swagger comes in. The goal of Swagger is to define a standard, language-agnostic interface for REST APIs, which allow humans and computers alike to discover and understand the capabilities of a given service.

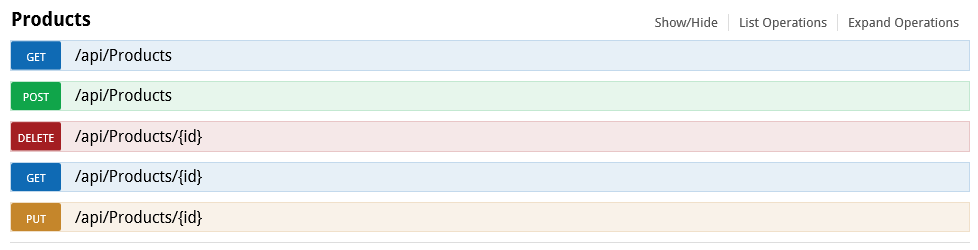
1. By default, the API App projects are enabled with automatic Swagger metadata generation. They include a Nuget package called Swashbuckle.Core, which handles the metadata generation for the WebAPI controllers and action methods inside the application.
2. This Nuget package comes with a UI page that allows humans easily discover and explore the capabilities of the services exposed in the application. We will work on enabling this page so that we can see the services offered by our application. From the Solution Explorer, open the App\_Start/SwaggerConfig.cs file.
3. In the code file, search for EnableSwaggerUI. Once you have located the code, uncomment it and save the file.



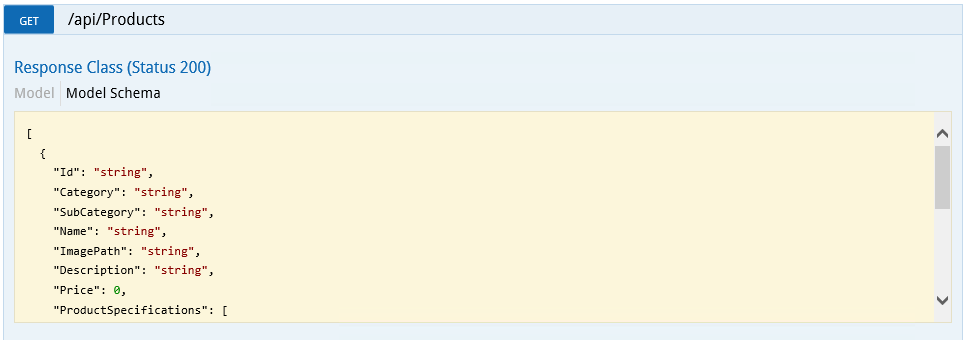
1. To run and start the application locally again, press Ctrl+F5. As a result, the Internet Explorer browser window will be displayed with the message "403 - access denied error message". This is normal, since / (slash) is not a valid WebAPI controller name – action method pair. Append **/swagger** at the end of the URL in the browser to go to the Swagger UI page.



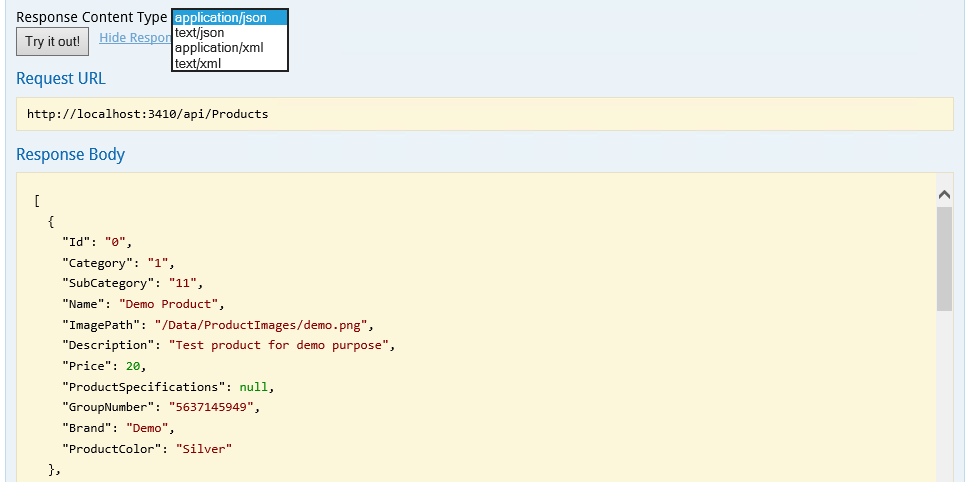
1. The Swagger page lists the names of all the WebAPI controllers that are present inside the application that we’ve coded. Click the **Products** controller, and a list of operations (action methods) that are available on this controller, including create, read, update, and delete action methods, will be displayed.



1. To view the details of operation, click the **/api/Products** read operation. You can now inspect the schema of the Product object that will be returned. Also, you can see that the call will bring back an array of Product objects. Each Product object will also load the ProductSpecications array, which will be comprised of ProductSpecifications objects.



1. The Swagger UI page also allows you to try out the service by making a call to it. Clicking the **Try it out** button on the page underneath the Model schema will send a GET call to the URI of the controller and invoke the GetProducts action method. It will format the result in a readable style, and also allow you to do content negotiation to request JSON, XML, and so on.



1. Invoke the GetProducts action method and check how many products did you get back. Also check if these products have any ProductSpecification objects associated to them or try adding some objects by using the SwaggerUI. Note that if you enable the SwaggerUI page on your application and the application allows anonymous access, anyone accessing the application will be able to discover the services.

### Task 4: Setting Up CRUD Operations

1. In the Solution Explorer of the Visual Studio, open the ProductCsvController.cs file. How many action methods does this controller contain? What are they supposed to be doing?
2. Add the following code to the PostProduct action method replacing the initial content of the action method entirely (remember, in WebAPI, methods that start with Post are activated by POST requests and are used to add new data):  
      
   if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

if (string.IsNullOrWhiteSpace(productCsv))

{

return BadRequest("CSV text content is null or empty");

}

var properties = productCsv.Split(',', ';');

if (properties.Length <= 7)

{

return BadRequest("CSV entry for the product doesn't seem right (< 7 properties)");

}

// Name, Description, Band, ProductColor, Price, ImagePath, Category, SubCategory

// Mapping

var product = new Product

{

Name = properties[0],

Description = properties[1],

Brand = properties[2],

ProductColor = properties[3],

Price = decimal.Parse(properties[4]), //price,

ImagePath = properties[5],

Category = properties[6],

SubCategory = properties[7]

};

product.Id = Guid.NewGuid().ToString();

db.Products.Add(product);

try

{

db.SaveChanges();

}

catch (DbUpdateException)

{

if (ProductExists(product.Id))

{

return Conflict();

}

else

{

throw;

}

}

return CreatedAtRoute("DefaultApi", new { id = product.Id }, product);

The code above will perform several things. It will check if an input string has been provided; we cannot create a product from no values. It will then try and split the string into an array of string, using the commas and the semicolons as separators. Should there be less than seven values in the array, we will not be able to map them to a new product, and if this is the case, an error is returned. If not, the array values are mapped onto a new product entity and then the entity will be saved to the database.

1. How can you now test if the code is working? Could you use the Swagger UI to create a new product? Try using the following CSV string to add a new product to the catalogue:

Dji Phantom 3,This is the latest of the Phantom drones,Dji,Gray,799,/Data/ProductImages/dji.png,1,11

1. How would you retrieve the product knowing its ID? Create a new empty WebAPI controller. To do so, in **Solution Explorer**, right-click the **Controllers** folder, and in the shortcut menu, select **Add** > **Controller**. Call the class CsvListController, name the file **ProductsCsvListController.cs** to keep in line with the other controllers. Add the following code inside the controller:

private EFContext db = new EFContext();

[ResponseType(typeof(Product[]))]

public IHttpActionResult PostProducts(string productCsv)

{

if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

if (string.IsNullOrWhiteSpace(productCsv))

{

return BadRequest("CSV text contnet is null or empty");

}

var elementsCsv = productCsv.Split('|');

var resultingProducts = new List<Product>();

//loop through the elements and attempt to create a product for each new element

foreach (string csvElement in elementsCsv)

{

var properties = csvElement.Split(',', ';');

if (properties.Length <= 7)

{

return BadRequest("CSV entry for the procduct doesn't seems right (< 7 properties)");

}

// Name, Description, Band, ProductColor, Price, ImagePath, Category, SubCategory

// Mapping

var product = new Product

{

Name = properties[0],

Description = properties[1],

Brand = properties[2],

ProductColor = properties[3],

Price = decimal.Parse(properties[4]), //price,

ImagePath = properties[5],

Category = properties[6],

SubCategory = properties[7]

};

product.Id = Guid.NewGuid().ToString();

db.Products.Add(product);

try

{

db.SaveChanges();

}

catch (DbUpdateException)

{

if (ProductExists(product.Id))

{

return Conflict();

}

else

{

throw;

}

}

//add the product to the list of products to be returned

resultingProducts.Add(product);

}

return Ok(resultingProducts);

}

private bool ProductExists(string id)

{

return db.Products.Count(e => e.Id == id) > 0;

}

Finally bring in the following namespaces at the top of the class:

using System.Data.Entity.Infrastructure;  
using System.Web.Http;

using System.Web.Http.Description;

using WebCatalogue.Models;

This code works exactly as the one before, however instead of passing in a string comprised of 7 properties that will be mapped to one new product, we will be passing in a string that will contain a series of several new object property listings, separated by the | (vertical dash) character. The code will first split the string into an array where each entry contains a listing of seven product properties, and then will attempt to map each listing to a new product.

1. Run the project again, go to the Swagger UI page URL (/swagger), identify the new controller, and test it out. You can use the following string to test the controller's action method:

Dji Phantom,The original DJI Phantom Drone,Dji,Gray,399,/Data/ProductImages/djiPhantom.png,1,11|Dji Phantom 2 Vision,The DJI Phantom 2 equiped with the Zemuse stabiliser and camera,Dji,Gray,799,/Data/ProductImages/djiVision.png,1,11

1. Once you have inserted the two new products, how can you go about getting their identifiers? Can you then find a way to delete these products with the existing controllers?

### Task 5: Advanced Data Queries for Products

Returning the entire list of products through a GET call from the API App might not be the best solution to implement, since, when the catalogue is filled with product data, this will return a large amount of data to the consuming client. We can set up various ways of retrieving product data using filtering.

1. Open the ProductsController.cs file and locate the GetProducts action method that returns the entire list of products. At the end of this method, we will add another action method to allow us to return either products with or without pictures, with the following code:

public IQueryable<Product> GetProducts(bool havePictures)

{

//if the onlyProductsWPictures parameter is true, then return onlt

//those products that have an existing picture

if ((bool)havePictures)

return db.Products.Where(p => !String.IsNullOrEmpty(p.ImagePath));

else

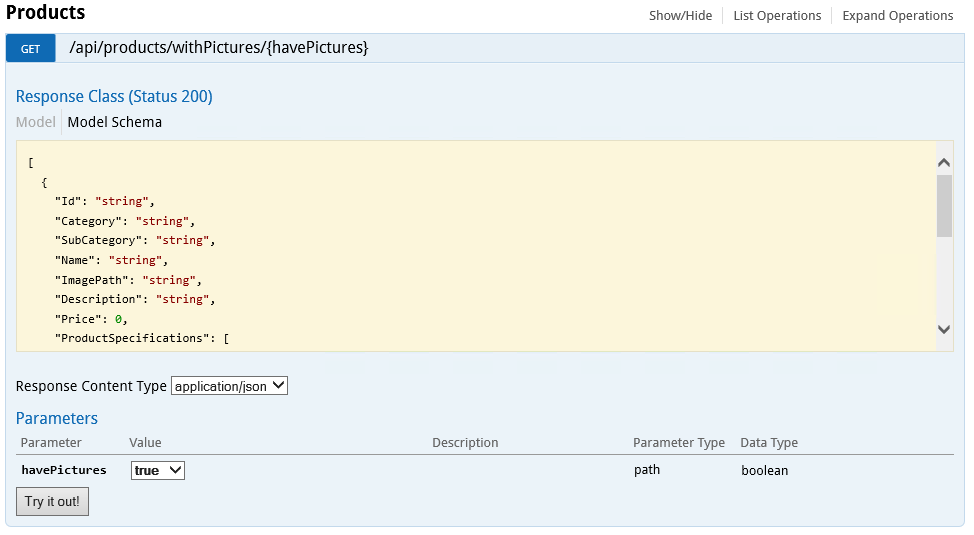
return db.Products.Where(p => String.IsNullOrEmpty(p.ImagePath));

}

1. Based on the value of the havePictures parameter, the action method will append a where clause to the the EntityFramework query so as to only return those products for which the ImagePath property has a value, or does not have a value.
2. If you run the solution as is, you will receive an error. What does the error indicate? What do you think is the source of the error?
3. In WebAPI, each controller implements action methods that follow an implicit convention (that can be overwritten with attributes if needed). Hence, methods that start with Get (GetProduct, GetProducts) will only be invoked on GET calls, action methods that start with Post (PostProduct) will only be invoked on POST requests.
4. The error you are seeing indicates that when a GET request is routed to this controller, WebAPI cannot decide which action method to invoke between GetPproduct(int) and GetProducts(bool). To differentiate, we can make use of attribute routing.
5. Add the following attribute before the action method definition we have just added:

[Route("api/products/withPictures/{havePictures:bool}")]

1. This instructs the application to only invoke this action when a route of type **/api/products/withPictures/false** or **/api/products/withPictures/true** is detected. Run the solution again and inspect the method in the Swagger UI.



1. Note that we clearly see that the route expects a parameter called havePictures at the end, which should be of type bool, as constrained in the routing attribute we have just added. Try invoking the method, and see how many products do not have pictures.
2. Create a product without an ImagePath value. You can use the Csv controller from the previous task to add a new product. Now run the method again, did you find products without pictures this time?

## Exercise 2: Deploying the solution

#### Objectives

In this exercise, we will be looking at the metadata properties of the application we have created in the previous exercise. Once we understand how these properties affect deployment, we will proceed to create an Azure API App through the Azure portal and deploy the application we have developed to the cloud.

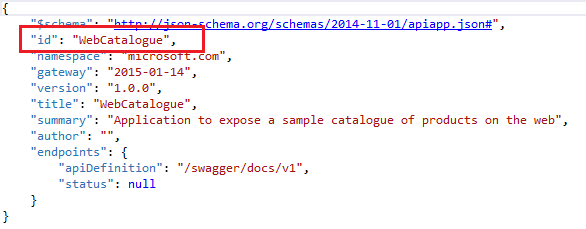
#### Scenario (if applicable)

Since we have completed testing the solution we have been building in Exercise 1, it is now time to go ahead with the deployment of this Azure API App to the Azure cloud, so we can make it accessible over the Internet.

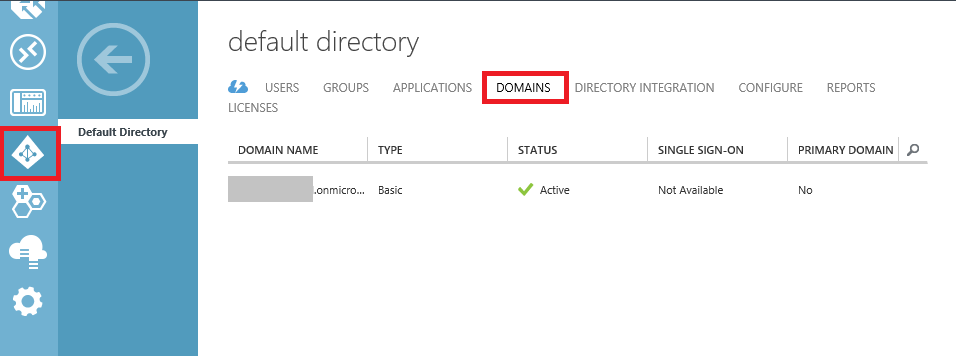
### Task 1: Configure Application Metadata

Most of the metadata properties of the application are to be found in the apiapp.json file and in the Metadata folder of the solution. The properties affect the way an API App package is presented in the Azure MarketPlace.

1. Set up the application ID. To do so, open the apiapp.json file and set the id to WebCatalogue. The id property determines the name of the API App.



1. The namespace metadata variable is used to indicate the domain name of Azure Active Directory tenant. To find the name of the domain, open the Azure [classic portal](http://azure.microsoft.com), then browse the Active Directory item, and select the **Domains** tab.



1. The endpoints metadata variable indicates the URL of the dynamic Swagger API definition. This URL is accessed through the GET request and will return the API definition in JSON format when the URL in the **endpoints.apiDefition** is called.

"endpoints": {

"apiDefinition": "/swagger/docs/v1"

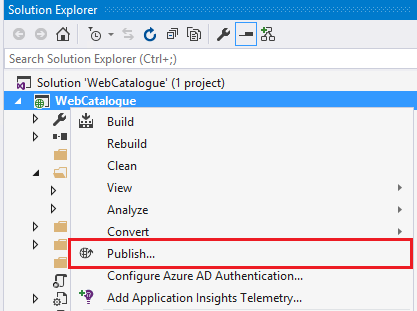
}

1. If you want to provide a static definition file, you can do so by storing the definition file inside the Metadata folder of the solution and naming the file apiDefinition.swagger.json.  
     
   Should you use static definition files inside your application, you should leave the endpoints.apiDefintion out of the apiapp.json or set its value to null. If you include both a static definition file (apiDefinition.swagger.json) and the URL, the URL inside the apiapp.json file will take precedence and the static definition file will be ignored.

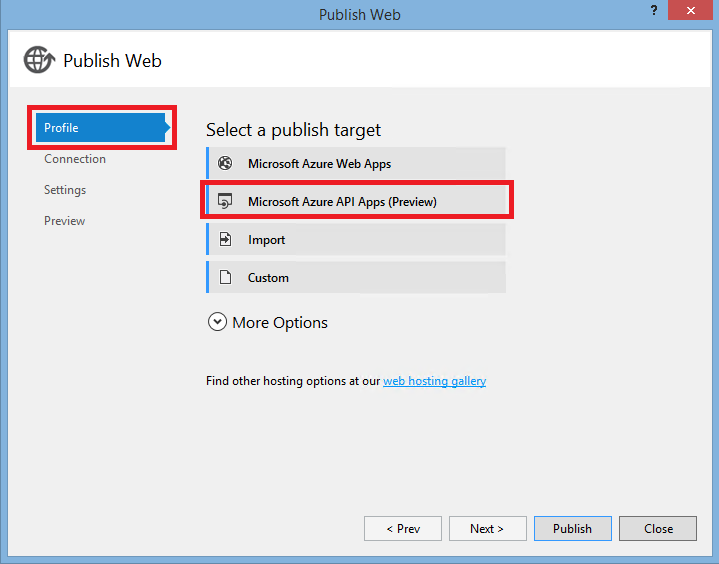
### Task 2: Create the Application in Azure

Follow the steps outlined below to create an Azure App instance in Azure and then deploy the code of the application we have created to the newly created application instance in the cloud.

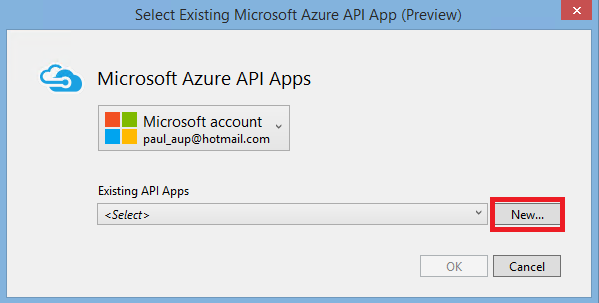
1. In Solution Explorer of Visual Studio 2015, right-click the **WebCatalogue** solution node and then from the shortcut menu, select **Publish**.

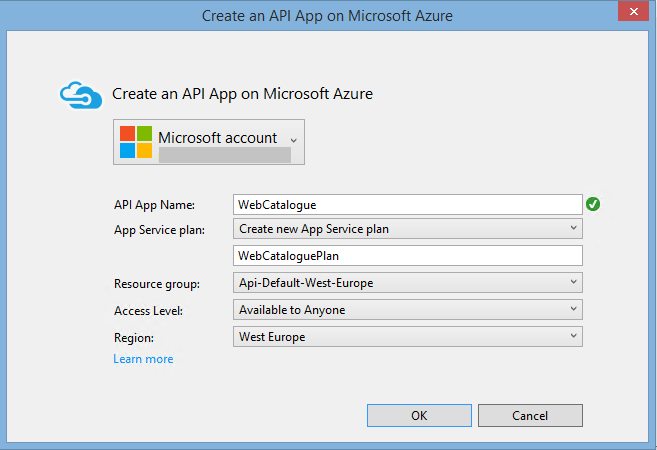


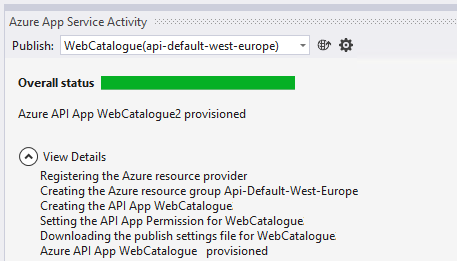
1. In the **Publish Web** dialog box, click the **Profile** tab and then click **Microsoft Azure API Apps (Preview)**.



1. In the **Select Existing Microsoft Azure API App (Preview)** dialog box, click the **New** button to provision a new instance of API App in your Azure subscription.



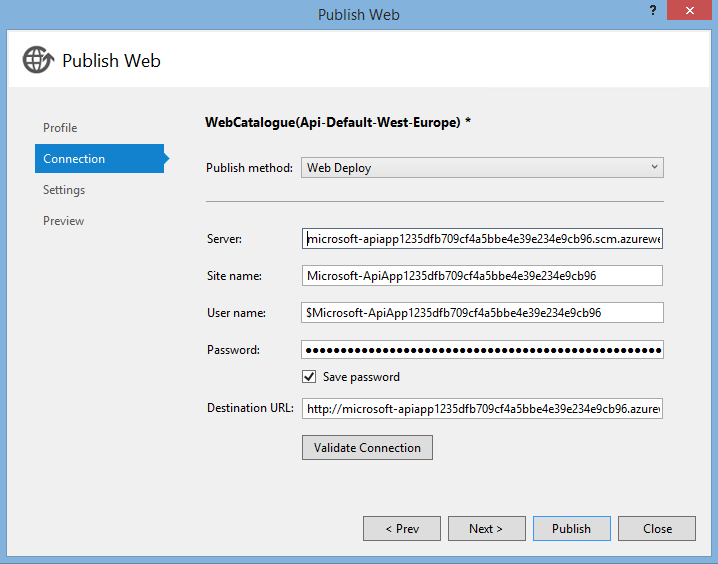
1. In the **Create an API App** dialog box, fill in the following information:
   1. If you have multiple Microsoft Azure subscriptions, chose the subscription you would like to use from the drop-down control.
   2. In the **API App Name** box, enter the name WebCatalogue that you have used for the application you have created.
   3. In the **App Service plan** box, create a new App service plan by entering the name of the new plan as **WebCataloguePlan**.
   4. In the **Resource group** box, select from one of your existing resource groups or create a new resource group.
   5. In the **Access Level** box, select the value to **Available to Anyone**. This will make the application available to anyone online. You can later change this through the Azure Preview portal.
   6. In the **Region** box, select the region that is closest to your physical location. If this were a real deployment, we would select the location that is closest to the majority of clients that will consume this service.
2. Click **OK** to have Visual Studio create the API App in your subscription. This process can take several minutes.
3. The provisioning process creates a new resource group and API App in your Azure subscription. Visual Studio shows the progress in the Azure App Service Activity window shown below.



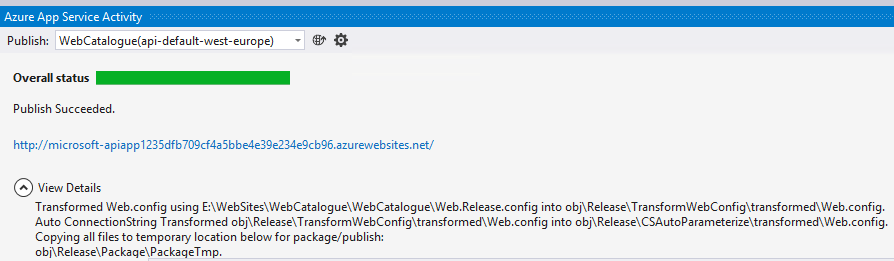
### Task 3: Deploy the Application's Code to Azure

Once the application has been correctly created in Microsoft Azure, you will need to execute the following steps to deploy the code (containing the business logic of the WebCAtalogue service) to the newly created Azure API App instance.

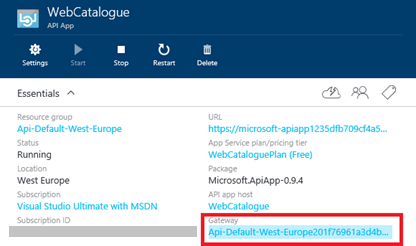
1. In Solution Explorer of Visual Studio 2015, right-click the **WebCatalogue** solution and from the shortcut menu, select **Publish**.
2. When the **Publish** window, the profile that you have created when following the steps outlined in Task 2 should have already been pre-selected by Visual Studio.



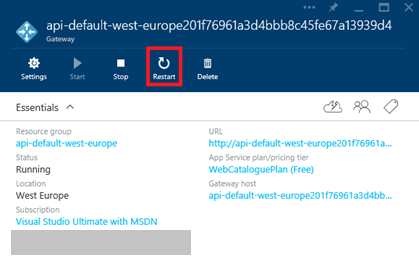
1. Pressing the **Publish** button will have Visual Studio attempt to deploy the code of the solution to the Azure App instance in the Azure cloud. As per the previous task, the Azure App Service Activity window will again show the progress Visual Studio is making to accomplish the deployment.



1. During the deployment process, Visual Studio will automatically try to restart the gateway. The gateway is a webapp that handles administrative functions for API Apps in a resource group, and has to be restarted to recognize changes in an API App's API definition or the apiapp.json file. If you use another method to deploy the API App, or Visual Studio fails to restart the gateway, you might have to restart the gateway manually. The next steps show you how to accomplish this.
2. Select the API App (WebCatalogue) from the list of API Apps that are available to you in the Azure Management portal. Clicking on WebCatalogue will bring up the API App details inside the portal as shown. Click the gatway name to access the gateway setting page.



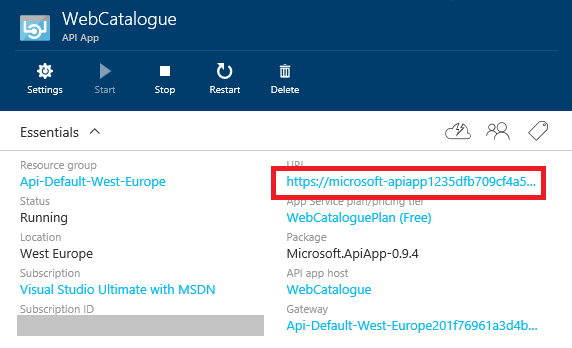
1. On the gateway settings page, click the **Restart** button to restart the gateway.



### Task 4: Access the Newly Created Azure API App

After the successful creation of the Azure API App and the deployment of the code from Visual Studio into the Azure cloud, the last step is to check if the API App actually works as expected when running in Azure. To do this, follow these steps:

1. In the Azure portal, go to the API App that you have deployed, and click the URL of the API App as shown.



A page indicating that the API App has been successfully deployed will load in your browser.

1. Add **/swagger** to the end of the URL to access the Swagger UI page and have access to the listings of all controllers and the action methods. Are you able to load the product list? Can you create a new product using a CSV string?

# Lab 2: Migrating an Existing WebAPI App

#### Introduction

The purpose of this lab is to show you that Microsoft Azure API Apps can be obtained by converting an existing WebAPI application that you might have on-premise on running in the cloud on a virtual machine into an Azure API App. During the lab, we will create a new WebAPI application to handle product picture operations, then convert this on-premise application in an Azure API App*.*

#### Objectives

After completing this lab, you will be able to:

* Know how to create a WebAPI application.
* Explain what the differences between a WebAPI and an Azure API App are.
* Convert and deploy an existing WebAPI App into an Azure API App.

#### Prerequisite

Lab M03 – Azure SQL Database : The **ProductsDB** database will be reused in this lab.

#### Estimated time to complete this lab

30 minutes

#### Scenario

To continue the work of exposing the catalogue of products that our company sells online through API calls hosted in Azure, we can imagine that we already have an existing WebAPI application that was hosted on-premise, and that allowed other applications on our Intranet to access or upload pictures associated with various products. We would like to migrate this existing application to become an Azure API App without having to create a new solution and move the code over one class at a time.

To do this, we will look at the following:

* How to create a WebAPI application with Visual Studio 2015 and WebAPI 2.
* Put in code that allows a controller to access Blob Storage in the Azure cloud and retrieve an image.
* Write code that allows someone to upload an image to the Azure cloud Blob Storage.
* Go through the phases of converting this application into an Azure API App that we will then deploy alongside the existing application created in the previous lab.

## Exercise 1: Creating the WebAPI application

#### Objectives

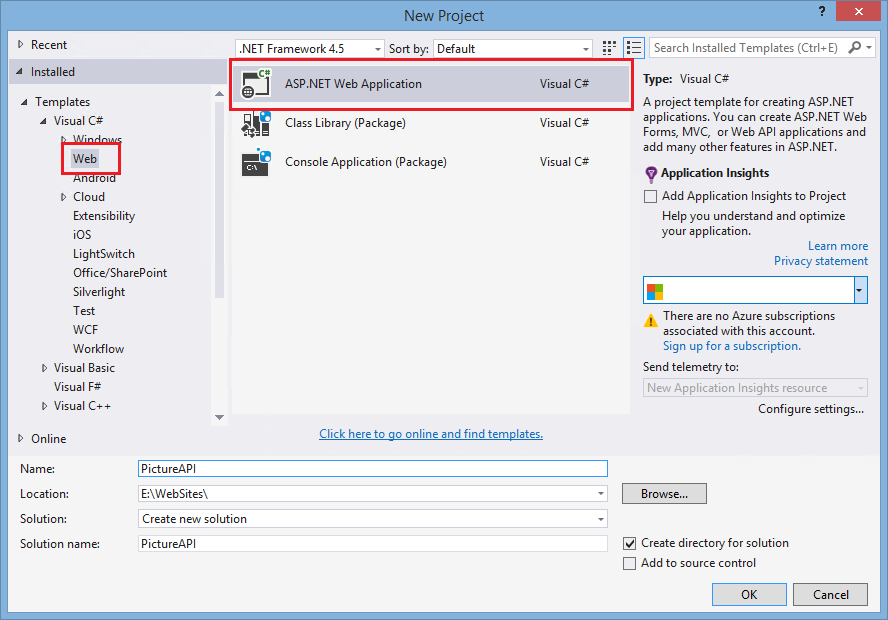
In this exercise, we will be creating a new WebAPI application using one of the built-in templates that comes with Visual Studio 2015 and WebAPI 2. We will then proceed to add the connection to the SQL Server Azure database and the business logic layer code to expose the data relevant to the pictures that may be associated with a product, and finally, write the code allowing us to retrieve pictures from Blob Storage or place pictures in Blob Storage.

#### Scenario

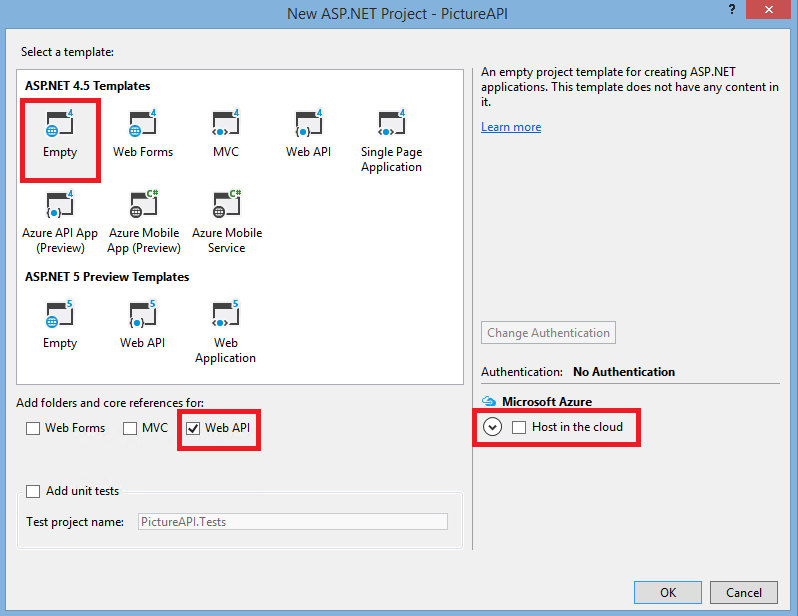
In order to complete the product catalogue listings, users in our company must have the ability to upload and associate images to products in the catalogue, or view the images that are already associated with a product. To do this, we will build a WebAPI application that will interact with the SQL Server Azure database to retrieve product information and with Azure Blob Storage to read and write the image data.

### Task 1: Creating the New WebAPI App

1. Open Visual Studio 2015, and click **File** > **New** > **Project**.
2. In the **New Project** dialog box, in the left pane, expand **Templates** > **Visual C#** and then click the **Web** category.



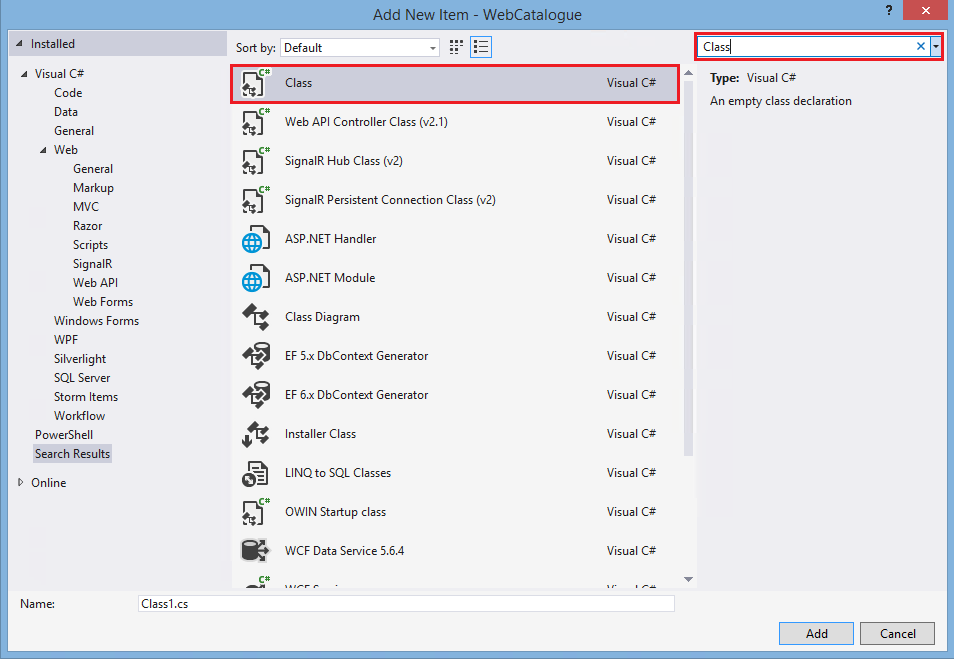
1. In the **New Project** dialog box, in the middle pane, click the **ASP.NET Web Application** project template.
2. In the **New Project** dialog box, in the **Name** box, enter the project **PictureAPI** and in the **Location** box, select a folder on your local machine where the solution files are to be created and stored by Visual Studio.
3. In the **New ASP.NET Project – PictureAPI** dialog box, under **ASP.NET 4.5 Templates**, click the **Empty** template. Then make sure that you select the **Web API** check box and clear the **Host in the cloud** check box, since initially this application is scheduled to work on-premise.



1. Click **OK** for Visual Studio to start generating the solution and its corresponding files.

### Task 2: Adding the Code to Connect to SQL Server Azure and Retrieve Products

1. In the solution you have just created, right-click the **Models** folder and in the shortcut menu, click **Add** > **New Item**.
2. In the **Add New Item – WebCatalogue** dialog box, in the search box, type the word **class** to find the class templates. In the search results displayed in the middle pane, select the **Class** item and make sure that you have selected a C# class and not a VB.NET.



1. In the **Add New Item – WebCatalogue** dialog box, in the Name box, enter the class name **DataModel.cs** and then add the following code inside the newly created file replacing the DataModel class definition created by Visual Studio 2015:

public class Product

{

public string Id { get; set; }

public string Category { get; set; }

public string SubCategory { get; set; }

public string Name { get; set; }

public string ImagePath { get; set; }

public string Description { get; set; }

public decimal Price { get; set; }

public List<ProductSpecification> ProductSpecifications { get; set; }

public string GroupNumber { get; set; }

public string Brand { get; set; }

public string ProductColor { get; set; }

}

public class ProductSpecification

{

public int Id { get; set; }

public string Name { get; set; }

public string Value { get; set; }

public bool? AllowComparision { get; set; }

public bool? AllowFiltering { get; set; }

}

This will create the entities for manipulating products and their associated product specifications objects.

1. Right-click the **Models** folder again, and in the shortcut menu, click **Add** > **New Item**. Add a new class like last time, and name the class **DataLayer.cs**.
2. Add the following code into the newly created file, replacing the existing definition of DataLayer:

public class MyDbInitializer : CreateDatabaseIfNotExists<EFContext>

{

public override void InitializeDatabase(EFContext context)

{

// Add command timeout 600

context.Database.CommandTimeout = 600;

base.InitializeDatabase(context);

}

}

public class SQLAzureConfiguration : DbConfiguration

{

public SQLAzureConfiguration()

{

this.SetExecutionStrategy("System.Data.SqlClient", () => new SqlAzureExecutionStrategy());

}

}

[DbConfigurationType(typeof(SQLAzureConfiguration))]

public class EFContext : DbContext

{

public EFContext() : base("DBConnection") { }

public DbSet<Product> Products { get; set; }

public DbSet<ProductSpecification> ProductSpecifications { get; set; }

}

This code will not compile, and Visual Studio will underline in red most of the lines inside the file. This is because we are missing the references to the Entity Framework ORM and the namespaces that Visual Studio needs to resolve the references to the objects we are using.

1. In the Tools menu in Visual Studio, bring up **Nuget Package Manager** > **Manage Nuget Packages for Solution**. Search for Entity Framework in the Package Manager and install the Nuget Package into your solution.
2. Finally, add the following two namespaces at the top of this file to allow Visual Studio to resolve references to objects in the code:

using System.Data.Entity;

using System.Data.Entity.SqlServer;

1. To complete the solution's data layer, add the following connection string to the connectionStrings section of the web.config file to be able to connect to the ProductsDB database in SQL Server Azure.

<add name="DBConnection" providerName="System.Data.SqlClient" connectionString="Server=tcp:{your\_server\_name}.database.windows.net,1433;Database=ProductsDB;User ID=ProductsDBAdmin@{your\_server\_name};Password={your\_password\_here};Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;" />

1. If the connectionStrings strings section does not exist, add the connection string to the web.config file after the appSettings section.

### Task 3: Adding the Image Upload / Download-Related Helper Classes

1. In the solution, right-click the **Models** folder, and from the shortcut menu, select **Add** > **New Item**.
2. In the **Add New Item** dialog box, search for class, and click to add a new C# class to the project. Name it **BlobModel.cs**. This class will contain the model classes for uploading and downloading the product-related images.
3. Add the following code replacing the definition of the BlobModel class:

public class BlobUploadedImageModel

{

public string FileName { get; set; }

public string FileUrl { get; set; }

public long FileSizeInBytes { get; set; }

public long FileSizeInKb {

get {

return (long)Math.Ceiling((double)FileSizeInBytes /

1024);

}

}

}

public class BlobDownloadImageModel

{

public MemoryStream BlobStream { get; set; }

public string BlobFileName { get; set; }

public string BlobContentType { get; set; }

public long BlobLength { get; set; }

}

1. Finally bring in the following namaspaces at the top of the file:

using System.IO;

1. In Solution Explorer, right-click the PictureAPI project node, and from the shortcut menu, select **Add** > **New Folder**. Name the new folder **Utilities**. This folder will be used to contain utility classes that will perform operations helping the solution upload and download blobs from Blob Storage.
2. In the Utilities folder, create a new C# class, and name the class **Blob Helper**. Once created, add the following code to the class:

public static CloudBlobContainer GetBlobContainer()

{

// Pull these from config

var blobStorageConnectionString =

WebConfigurationManager.AppSettings["BlobStorageConnectionString"];

var blobStorageContainerName =

WebConfigurationManager.AppSettings["BlobStorageContainerName"];

// Create blob client and return reference to the container

var blobStorageAccount =

CloudStorageAccount.Parse(blobStorageConnectionString);

var blobClient = blobStorageAccount.CreateCloudBlobClient();

var container =

blobClient.GetContainerReference(blobStorageContainerName);

//create the container if it does not exist - and set its access to blob,

//which means users can download a blob given they know the URL, but cannot

//list the contents of the container without auth

container.CreateIfNotExists(BlobContainerPublicAccessType.Blob,

null, null);

return container;

}

1. Add the Microsoft Azure Storage Nuget package to the solution and bring in the following namespaces:

using Microsoft.WindowsAzure.Storage;

using Microsoft.WindowsAzure.Storage.Blob;

using System.Web.Configuration;

This code will read from the application's web.config section and get two variables: **BlobStorageConnectionString**, which will contain the connection information to connect to Blob Storage and **BlobStorageContainerName**, which will indicate the container name to be used for operations.  
**Note** If the container does not exist, a new container with the name specified in the configuration will be created.

1. Create a second C# class in the **Utilities** folder and call this class BlobStorageUploadProvider. Then add the following code to the class:

public class BlobStorageUploadProvider : MultipartFileStreamProvider

{

public BlobUploadedImageModel UploadedImage { get; set; }

private string \_blobFileName;

public BlobStorageUploadProvider(String blobFileName) :   
 base(Path.GetTempPath())

{

//intialize the UploadedImage property to a new empty object

UploadedImage = new BlobUploadedImageModel();

//save the name of the blob file that is passed into the constructor

\_blobFileName = blobFileName;

}

public override Task ExecutePostProcessingAsync()

{

//FileData is a property of MultipartFileStreamProvider and is a list of multipart

//files that have been uploaded and saved to disk in the Path.GetTempPath()

//location we are only interested in the first uploaded file,

//should there be more than one

var fileData = FileData.First();

//Sometimes the filename has a leading and trailing double-quote character

//when uploaded, so we trim it; otherwise, we get an illegal character exception

string fileName =   
 Path.GetFileName(fileData.Headers.ContentDisposition.FileName.Trim('"'));

//Retrieve reference to a blob

var blobContainer = BlobHelper.GetBlobContainer();

var blob = blobContainer.GetBlockBlobReference(\_blobFileName);

//set the content type of the new blob

blob.Properties.ContentType = fileData.Headers.ContentType.MediaType;

//copy from local storage into the Windows Azure blob service

using (var fs = File.OpenRead(fileData.LocalFileName))

{

blob.UploadFromStream(fs);

}

//delete the local file

File.Delete(fileData.LocalFileName);

//create blob upload model with properties from blob info

var blobUpload = new BlobUploadedImageModel

{

FileName = blob.Name,

FileUrl = blob.Uri.AbsoluteUri,

FileSizeInBytes = blob.Properties.Length

};

//set the reference to the uploaded blob metadata to the upload property

UploadedImage = blobUpload;

return base.ExecutePostProcessingAsync();

}

}

1. Bring in the following namespaces:

using PictureAPI.Models;

using System.Net.Http;

using System.IO;

using System.Threading.Tasks;

1. The class inherits from the MultiPartStreamProvider class in System.Net.Http. The constructor of the class allows the caller to specify the name of the desired blob. The class then overwrites the ExcecutePostProcessingAsync method of the base class, providing the code to logic to retrieve the uploaded file from the request, then upload the file to BlobStorage.
2. Finally, in the web.config of the application, add the two key / value pairs to specify the connection string to Blob Storage and the name of the container to be used. Open web.config and in the **<AppSettings></AppSettings>** section, add the following:

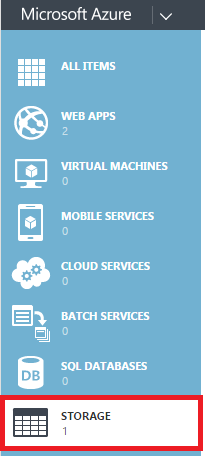
<appSettings>

<add key="BlobStorageConnectionString"   
value="DefaultEndpointsProtocol=https;AccountName={yourAccountName};AccountKey={yourAccountKey}" />

<add key="BlobStorageContainerName" value="webcatalogue" />

</appSettings>

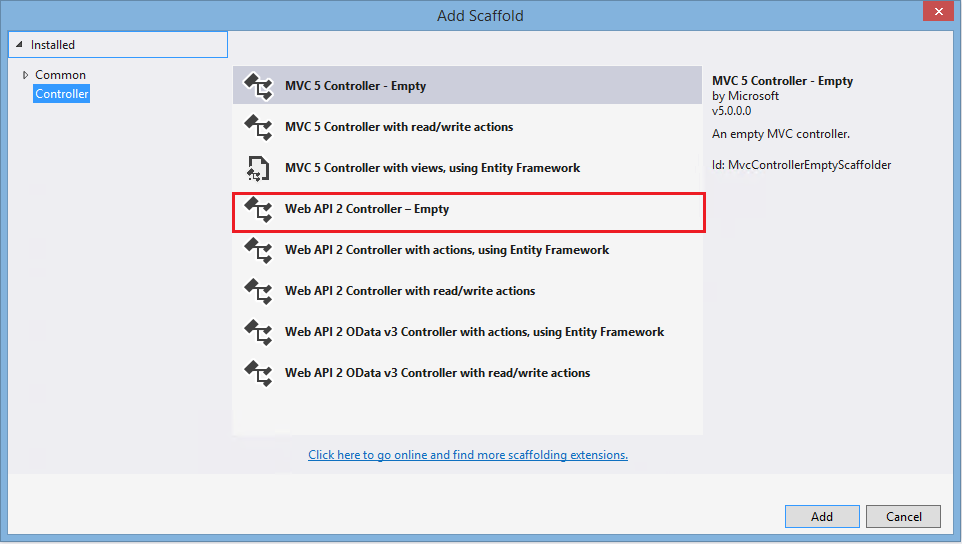
1. From the Management portal, retrieve the credentials to the storage account you will use for the application by clicking **STORAGE** in the portal and then selecting the storage you want to use.



1. Once you select the storage that you want to use, click the **Manage Access Keys** button located at the bottom bar of the portal to retrieve the **Storage Account Name** and **Primary Access Key** values, which you should replace in web.config of the application.

**Task 4: Adding the Controller Logic to Upload and Download Images**

1. To complete the solution, we will add the WebAPI controller allowing the upload and download of images associated to products from Blob Storage. In Solution Explorer of Visual Studio, right-click the **Controllers** folder, and from the shortcut menu, select **Add** > **Controller**.
2. Select to add a new empty WebAPI 2 controller from the scaffolding wizard that is displayed, as shown below:

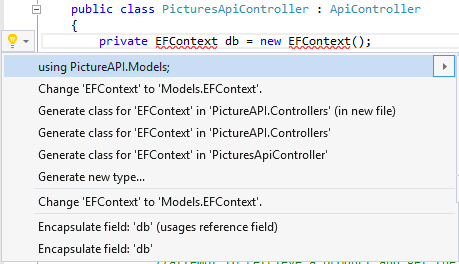


1. Name the controller **PictureAPIController** and complete the wizard.
2. In the new controller, add the following line of code before the code of the action method defined inside:

private EFContext db = new EFContext();

This will create a new Entity Framework context that the controller will use to query the SQL Server Azure database for products and their related images.

**Note** To bring in any missing namespaces, place the cursor on text underlined in red, then press Ctrl+. (dot) to have Visual Studio automatically bring up the possibilities for namespace import, as shown in the image below:



1. Now replace the empty action method created by the scaffolding wizard with the following action method to download product pictures:

[Route("api/{id}/picture")]

public async Task<HttpResponseMessage> GetPicture(string id)

{

// get a hold of the product and check if it has an associated image

Product product = db.Products.Find(id);

//should there be no such products or the image path be empty - return null

if (product == null || String.IsNullOrWhiteSpace(product.ImagePath))

{

return new HttpResponseMessage(HttpStatusCode.NotFound);

}

else

{

try

{

var blobName = product.ImagePath;

//should the name begin wih a slah, remove the first slash

if (blobName.StartsWith("/"))

{

blobName = blobName.Substring(1);

}

var container = BlobHelper.GetBlobContainer();

var blob = container.GetBlockBlobReference(blobName);

//download the blob into a memory stream. Notice that we're not putting the   
 //memory stream in a using statement. This is because we need the stream to be   
 //open for the API controller in order for the file to actually be downloadable.  
 //The closing and disposing of the stream is handled by the Web API framework.

var ms = new MemoryStream();

await blob.DownloadToStreamAsync(ms);

//strip off any folder structure so the file name is just the file name

var lastPos = blob.Name.LastIndexOf('/');

var fileName = blob.Name.Substring(lastPos + 1,   
 blob.Name.Length - lastPos - 1);

// Build and return the download model with the blob stream and its relevant info

var download = new BlobDownloadImageModel

{

BlobStream = ms,

BlobFileName = fileName,

BlobLength = blob.Properties.Length,

BlobContentType = blob.Properties.ContentType

};

//Reset the stream position; otherwise, download will not work

download.BlobStream.Position = 0;

//create response message with blob stream as its content

var message = new HttpResponseMessage(HttpStatusCode.OK)

{

Content = new StreamContent(download.BlobStream)

};

// Set content headers

message.Content.Headers.ContentLength = download.BlobLength;

message.Content.Headers.ContentType =

new MediaTypeHeaderValue(download.BlobContentType);

message.Content.Headers.ContentDisposition =

new ContentDispositionHeaderValue("attachment")

{

FileName = HttpUtility.UrlDecode(download.BlobFileName),

Size = download.BlobLength

};

//return the model image object

return message;

}

catch (Exception ex)

{

return new HttpResponseMessage

{

StatusCode = HttpStatusCode.InternalServerError,

Content = new StringContent(ex.Message)

};

}

}

}

This method will try to find a product with the id that was passed in from the route. If it does, it will then look at the ImagePath property of the product object and check if a value is provided. If one is present, it will then proceed to connect to blob storage, download the blob, and the stream the contents to the response.

1. Add the following action method to the controller to allow a picture to be uploaded to a given product:

[Route("api/uploadpicture/{id}")]

[ResponseType(typeof(BlobUploadedImageModel))]

public async Task<IHttpActionResult> PostPicture(string id)

{

//check if the request we are receiving is of type multipart-content

if (!Request.Content.IsMimeMultipartContent("form-data"))

{

return StatusCode(HttpStatusCode.UnsupportedMediaType);

}

//attempt to retrieve a product and get the image path from it to be passed to

//the blob storage provider

Product product = db.Products.Find(id);

if (product == null ||

String.IsNullOrWhiteSpace(product.ImagePath))

{

return NotFound();

}

else

{

try

{

//create the blob upload storage provider to help with the upload to blob storage

var blobUploadProvider =

new BlobStorageUploadProvider(product.ImagePath);

var uploadResult =

await Request.Content.ReadAsMultipartAsync(blobUploadProvider)

.ContinueWith(task =>

{

if (task.IsFaulted || task.IsCanceled)

{

throw task.Exception;

}

var provider = task.Result;

return provider.UploadedImage;

});

if (uploadResult != null)

{

return Ok(uploadResult);

}

// Otherwise

return BadRequest();

}

catch (Exception ex)

{

return InternalServerError(ex);

}

}

}

This action method will attempt to find a product by the id that was provided in the route, and then retrieve the name of the image file it is supposed to have associated with it. It will then use the BlobStorageUploadProvider object to upload the blob with the name indicated by the ImagePath variable of the product object, to Azure BlobStorage.

1. Bring in the following namespaces:

using System.Threading.Tasks;

using PictureAPI.Utilities;

using System.IO;

using System.Net.Http.Headers;

using System.Web;

using System.Web.Http.Description;

1. Finally, in Solution Explorer of Visual Studio, right-click the solution node, and from the shortcut menu, select **Build Solution** to compile the newly created and complete solution.

**Task 5: Testing the Image Download with the WebAPI Solution**

1. Connect to BlobStorage using the Visual Studio Server Explorer, as shown in the Storage chapter. Select the same storage account as you have configured in the web.config of the solution you have just created.
2. Create a container called **webcatalogue** if you do not already have such a catalogue inside the storage account.
3. Using the solution build in the previous lab, get all products inside the WebCatalogue database from SQL Server Azure and select a product that has a value for the ImagePath variable. Note down the product ID (guid) and the value of the ImagePath variable.
4. Upload a picture to blob storage through Visual Studio and name the blob with the exact same name as the value you wrote down from the ImagePath variable of the product you selected in the previous step. Include any folder paths in the name (for example, if the name was ‘/data/images/demo.png’, the name of the blob should be ‘/data/images/demo.png’. Remember that Blob Storage is flat and folders are represented in the file names).
5. Start the PictureAPI solution from Visual Studio, by selecting **Run** from the **Test** menu. When the browser is displayed, type the following URL at the end of the existing address: /api/ProductGuid/picture – where ProductGuid was the value of the Guid for the product you selected in Step 3.

## Exercise 2: Converting the application to Azure API

#### Objectives

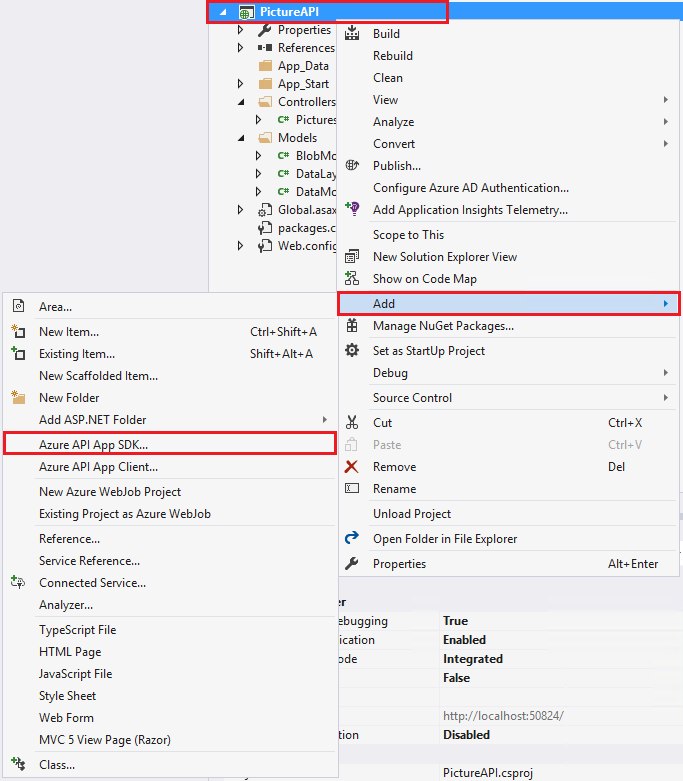
In many cases, you will be confronted with the migration of existing WebAPI solutions on-premise to Azure API to enable their hosting in the cloud. This exercise will walk you through how the existing WebAPI solution to upload and download product-related pictures can be converted to an Azure API application.

#### Scenario

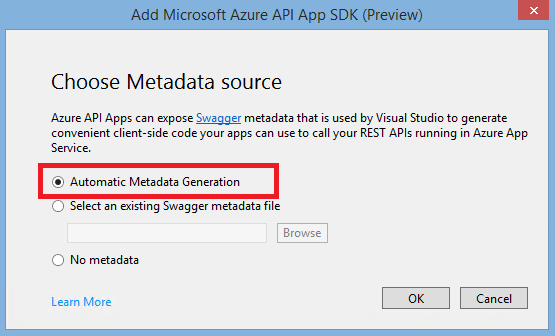
The existing application (PicturesAPI) that was built in the last exercise was very suitable to hosting in an on-premise datacenter. When envisaging a move to the Azure cloud, and allowing the solution to scale dynamically, the project will have to be converted to an Azure API App before it can be hosted in the Azure cloud.

### Task 1: Adding the Nuget Packages Needed for Conversion to Azure API

1. Open the completed WebAPI solution called PictureAPI from the last exercise. You should be able to find the solution in the **Start** folder of this lab.
2. In Solution Explorer of Visual Studio, right-click the **PictureAPI** project node, and from the shortcut menu, select **Add** > **Azure API App SDK**.

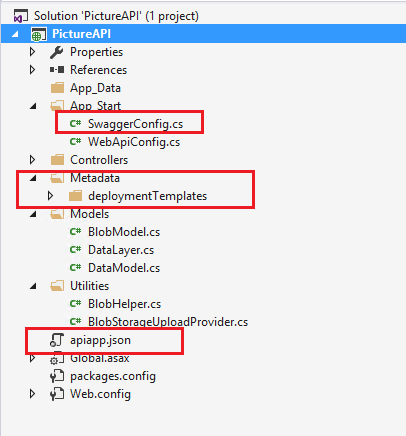


1. In the **Add Microsoft Azure API App SDK (Preview)** dialog box, click **Automatic Metadata Generation** using Swagger API.

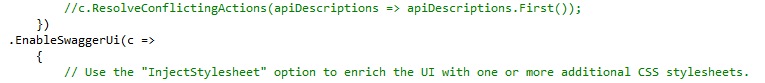


This option enabled the dynamic Swagger API, which was already displayed in the first lab of this Chapter.

1. Clicking OK will add the needed components to the solution effectively converting the existing WebAPI project into an Azure API App, by adding the necessary Nuget packages to the project.
2. Note that following the conversion, the Metadata folder, the API definition file, apiapp.json, and a class to configure the Swagger API have been added to the project and are visible in the Visual Studio Solution Explorer.



1. Open the SwaggerConfig.cs file located in the App\_Start folder of the solution. Locate the following lines of code and make sure they are uncommented (they should be by default):



1. Run the project, and when the browser comes up, add **/swagger** at the end of the address in the address bar of the browser to see that the Swagger UI is displayed as in the previous lab.