ASP.NET Core 2.1 with SQL Server Deployed to Azure

By Will Stott

This is a supplementary article that complements two feature articles published in the November and December 2108 editions of MSDN Magazine. The first feature, titled "Web Site Background Processing with Azure Service Bus Queues," was published in the November 2018 issue of MSDN Magazine. The second feature, to be published in the December 2018 issue, is titled "Using Azure Containers to Provide an On-Demand R Server."

These articles address the problems of providing a simple Web site with the ability to perform background processing using an on-demand server created from a Docker image running as an Azure Container. They address the key problems encountered in building a simple, low-budget Web site capable of demonstrating the logistic regression classifier I had developed as part of my research for the United Kingdom Collaborative Trial of Ovarian Cancer Screening (UKCTOCS). This article provides step-by-step instructions for building the base ASP.NET Core 2.1 Web App and SQL Database, as well guidance to publish them to corresponding resources provisioned on Azure.

The Web site is just simple a Web form that captures three integers representing ovary dimensions, saves them in a database record, and then displays a result page to show whether or not the classifier judges these dimensions to be consistent with those of an ovary. An overview of the Web App part of the system is shown in **Figure 1** and you can see the final product at **www.ovaryvis.org.**

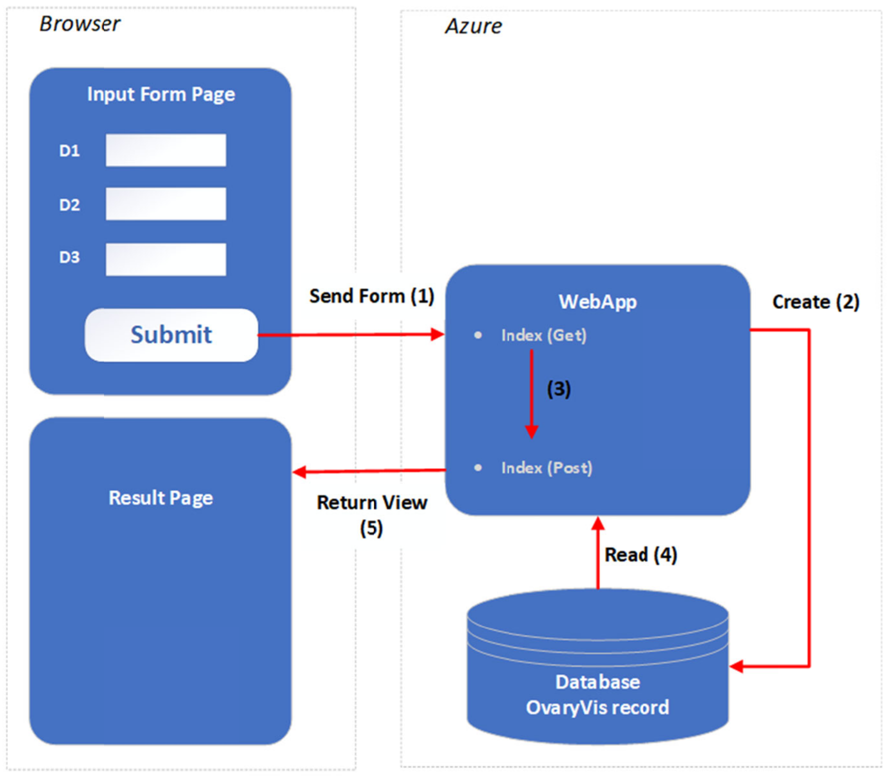


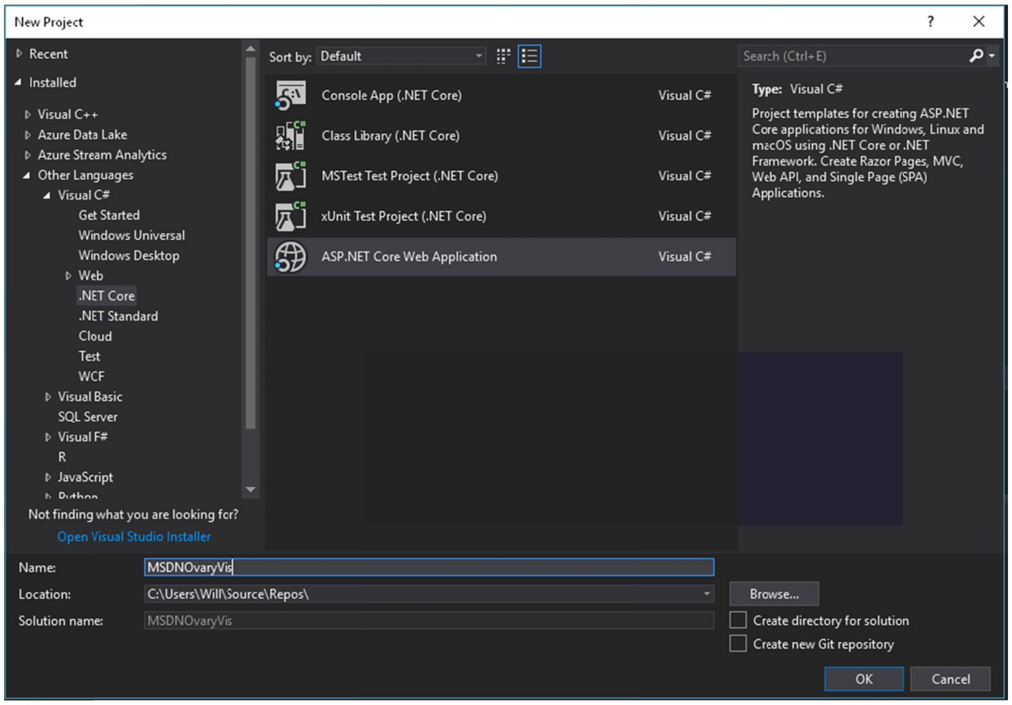
Figure 1 **System Overview**

To recreate the project featured in this article you shouldn't need much more than rudimentary Web development skills. In terms of tools, you'll need Visual Studio 2017 v15.7 with the Web development workload and .NET Core 2.1 SDK , as well as the SQL Server Data Tools (SSDT) However, the free community edition of Visual Studio should work fine too. In addition, you'll need an Azure subscription, but you can also get what you need for free if you're a new customer. All the source code and instructions for creating the Azure resources are available from the GitHub repository at **github.com/wpqs/MSDNOvaryVis**.

Building the Web App

Visual Studio provides a number of templates that allow you to build a Web App on your local development PC with a few mouse clicks. By the end of this section, you'll have created a Web App based on the ASP.NET Core 2.1 MVC framework that you can test using the local version of IIS installed on your PC with Visual Studio.

Creating an ASP.NET Core 2.1 MVC Web App is simply a matter of creating a new Visual Project (File | New Project), and selecting the ASP.NET Core Web Application template found in the Visual C# languages/.NET Core folder, as shown in **Figure 2**.


New Project
?
Recent
Installed
Visual C++
Azure Data Lake
Azure Stream Analytics
Other Languages
Visual C#
Get Started
Windows Universal
Windows Desktop
Web
.NET Core
.NET Standard
Cloud
Test
WCF
Visual Basic
SQL Server
Visual F#
R
JavaScript
Python
Not finding what you are looking for?
Open Visual Studio Installer
Sort by: Default
Search (Ctrl+E)
Console App (.NET Core) Visual C#
Class Library (.NET Core) Visual C#
MSTest Test Project (.NET Core) Visual C#
xUnit Test Project (.NET Core) Visual C#
ASP.NET Core Web Application Visual C#
Browse...
Create directory for solution
Create new Git repository
OK
Cancel

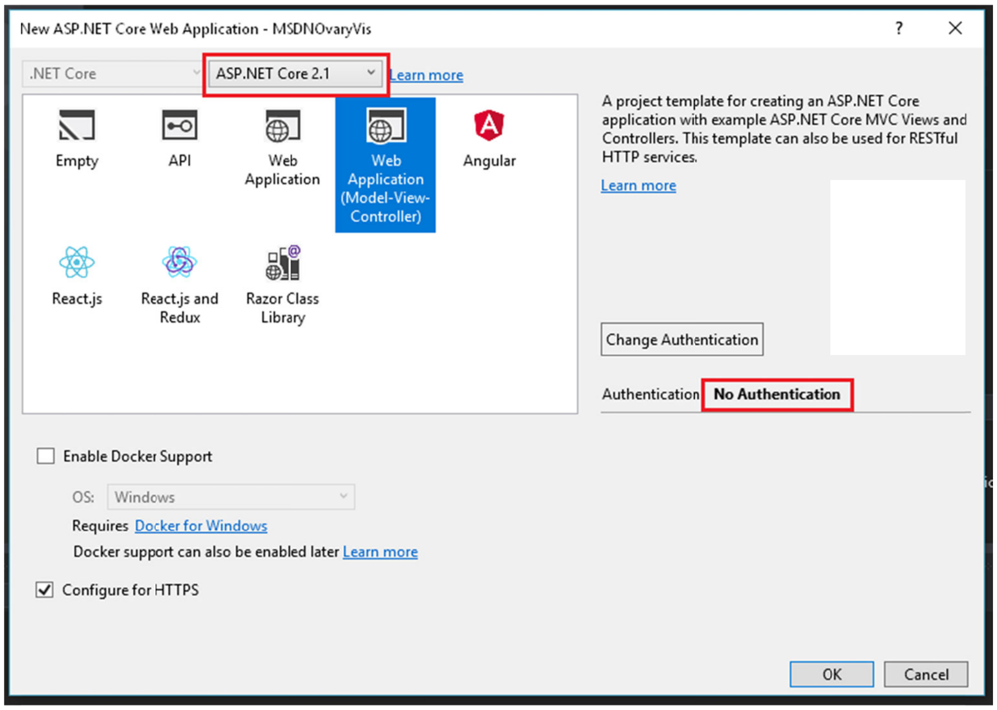
Type: Visual C#

Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework. Create Razor Pages, MVC, Web API, and Single Page (SPA) Applications.

Name: MSDNOvaryVis
Location: C:\Users\Will\Source\Repos\
Solution name: MSDNOvaryVis

Figure 2 **Visual Studio New Project Dialog**

You'll then need to select the Web Application (Model-View-Controller) in the option dialog box shown in Figure 3. Remember to select ASP.NET Core 2.1 as the target in the dropdown box at the top of the dialog box, and keep the default No Authentication setting. Visual Studio will then create all the files and settings you need for your Web App. There's a good recipe that takes you through this process on the Microsoft Docs site titled "Getting Started with EF Core on ASP.NET Core with a New database."


New ASP.NET Core Web Application - MSDNOvaryVis

.NET Core | ASP.NET Core 2.1 | Learn more

A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

Learn more

Change Authentication

Authentication: No Authentication

☐ Enable Docker Support

OS: Windows

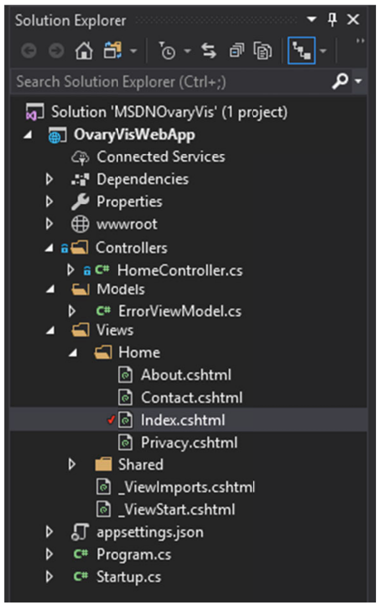
Requires Docker for Windows

Docker support can also be enabled later Learn more

☑ Configure for HTTPS

Figure 3 **Options for Creating a Web Application**

The naming of the folders in your initial Web App reflects the Model-View-Controller pattern, which is a well-known way of structuring applications. The Model folder contains classes that map to the tables in your database and provide the data for the Web site. The Views folders contain the HTML mark-up files needed to present the model data in the user's browser. Finally, the Controller folder contains the classes that provide the logic for constructing the views as requested by the browser's URL. You can see these folders and their contents in **Figure 4**.


Solution Explorer

Search Solution Explorer (Ctrl+;)

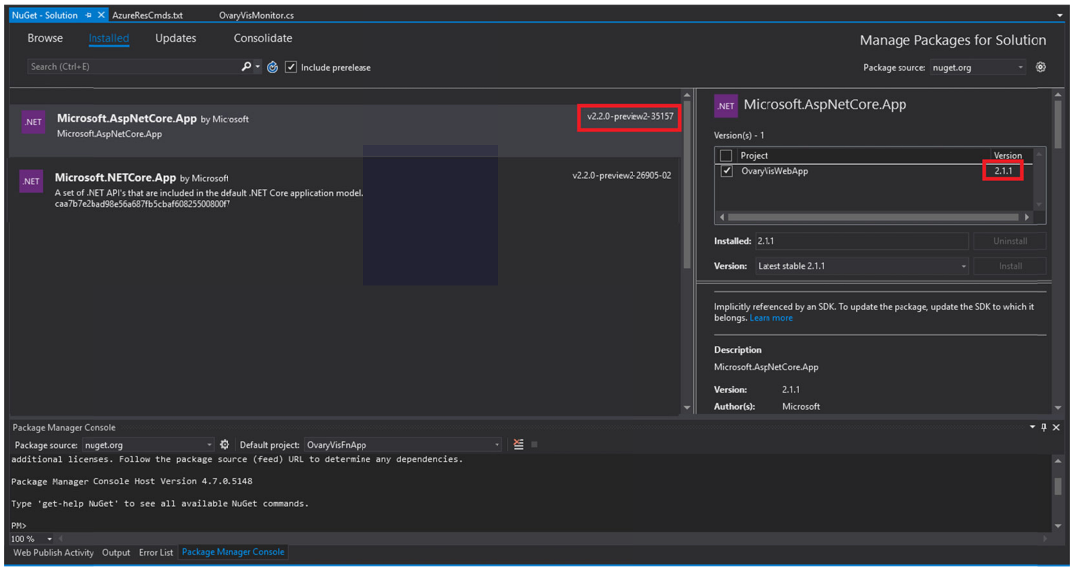
```
Solution 'MSDNOvaryVis' (1 project)
  OvaryVisWebApp
    Connected Services
    Dependencies
    Properties
    wwwroot
    Controllers
      HomeController.cs
    Models
      ErrorViewModel.cs
    Views
      Home
        About.cshtml
        Contact.cshtml
        Index.cshtml
        Privacy.cshtml
      Shared
      _ViewImports.cshtml
      _ViewStart.cshtml
    appsettings.json
    Program.cs
    Startup.cs
```

Figure 4 **Initial Folders and Files Created for the Web Application**

Microsoft follows a particular convention to route a given URL to the appropriate controller, which is defined in the Configure method of the Startup class. This means that a URL like **ovaryvis.org/Home/Result?Id=xyz** will get routed to the Result method (action) of the HomeController class with the query Id=xyz passed as a parameter, as you will see when you implement this action later.

The view returned by the Result method, by convention, will be the HTML file Result.cshtml, located in the Home subfolder of the Views folder. However, the controller may decide to return other views, say Error as found in the Views\Shared folder. Sticking to this convention makes for simpler code, though you can provide your own routing when required. You can learn more from "Routing in ASP.NET Core" on the Microsoft Docs site.

Building and testing your Web App just requires you to press Ctrl+F5 (Debug | Start without Debugging). You then need to wait for the home page to appear in your default browser so you can perform a quick smoke test to check that the menus and pages work as you anticipate. This is also a good time to check that you have the latest versions of the packages needed for the project as displayed in the NuGet – Solution window. You can open this window by clicking Tools | NuGet Package Manager | Manage NuGet Packages for Solution as shown in **Figure 5**. I used Microsoft.AspNetCore.App v2.1.1 and Microsoft.NETCore.App v2.1.1 for this article, but you might want to try later versions for your own work.



Microsoft.AspNetCore.App by Microsoft v2.2.0-preview2-35157
Microsoft.AspNetCore.App

Microsoft.NETCore.App by Microsoft v2.2.0-preview2-26905-02
A set of .NET API's that are included in the default .NET Core application model.
caa7b7e2bad98e56a687fb5cbaf60825500800f7

Microsoft.AspNetCore.App
Version(s) - 1

Project	Version
OvaryVisWebApp	2.1.1

Installed: 2.1.1
Version: Latest stable 2.1.1

Implicitly referenced by an SDK. To update the package, update the SDK to which it belongs. Learn more

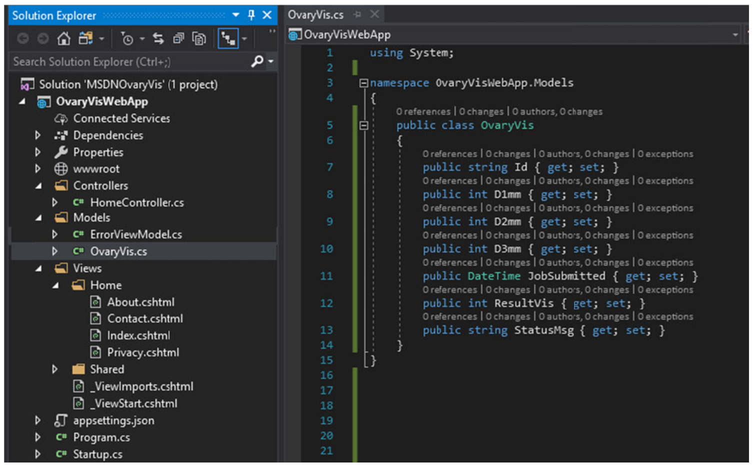
Description
Microsoft.AspNetCore.App
Version: 2.1.1
Author(s): Microsoft

Package source: nuget.org Default project: OvaryVisFnApp

```
additional licenses. Follow the package source (feed) URL to determine any dependencies.

Package Manager Console Host Version 4.7.0.5148

Type 'get-help NuGet' to see all available NuGet commands.

PM>
```

Figure 5 **The Visual Studio NuGet – Solution Window**

Creating the model class OvaryVis implements the basic unit of data that will be passed around the various components of your solution. It is simply a class containing a collection of properties that correspond to the fields in the database record we want to create, as shown in **Figure 6**. In this case the properties correspond to the dimensions of an ovary, as well as the resulting classification code where -1 = not yet classified, 0 = not visualized, 1 = visualized. There are also properties for the time the record was created and a status message so you can give feedback to users about the progress of processing.

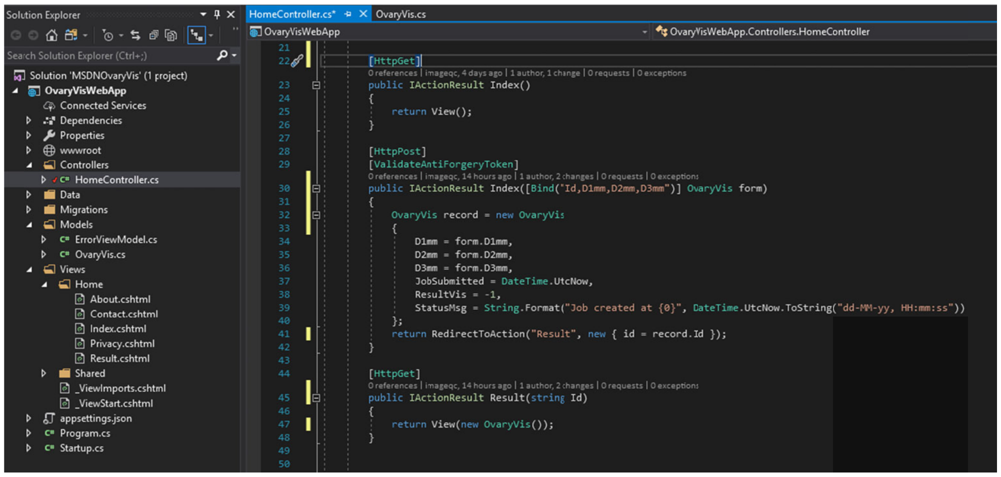


```
using System;

namespace OvaryVisWebApp.Models
{
    0 references | 0 changes | 0 authors, 0 changes
    public class OvaryVis
    {
        0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
        public string Id { get; set; }
        0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
        public int D1mm { get; set; }
        0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
        public int D2mm { get; set; }
        0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
        public int D3mm { get; set; }
        0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
        public DateTime JobSubmitted { get; set; }
        0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
        public int ResultVis { get; set; }
        0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
        public string StatusMsg { get; set; }
    }
}
```

Figure 6 **The OvaryVis Class in the Models Folder**

Implementing the Web App user interface (UI) requires you to construct views that allow the collection and display of the data in the model class OvaryVis. The routing convention mentioned above ensures that the Index method of the HomeController responsible for handling HttpGet requests is called when a user lands on your Web site's home page. This method creates the Index view page, which contains a standard HTML form allowing the user to enter the required input data. When the user clicks the submit button on the form, a Post message is generated by the browser containing the form data. This message is returned to your Web site where it is routed to the Index method of the HomeController responsible for handling HttpPost requests, together with a parameter containing the form data object. The Index method creates a new OvaryVis object and populates it with the form data before invoking the Result method, passing the object's Id as a parameter, as shown in **Figure 7**.

Clearly the Index method must somehow save the OvaryVis object so it can be found from its Id value in the Result method. This will require the database that I'll implement in the following section. For the moment, just to get things working, we'll create a new OvaryVis object in the Result method and use it to construct the Result View.



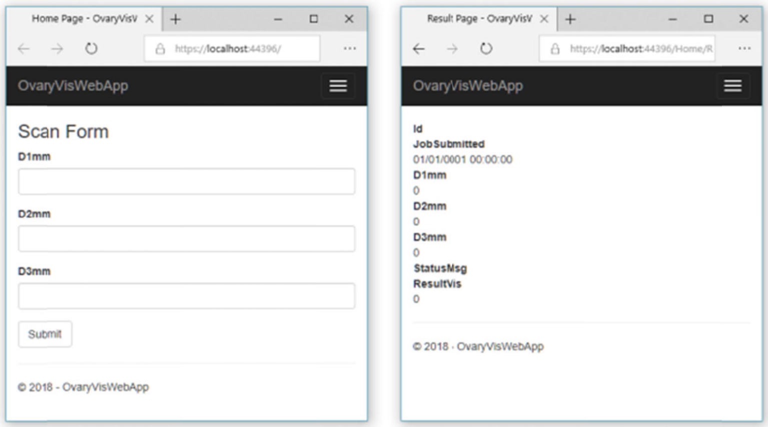
```

        [HttpGet]
        0 references | imageqc, 4 days ago | 1 author, 1 change | 0 requests | 0 exceptions
        public IActionResult Index()
        {
            return View();
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        0 references | imageqc, 14 hours ago | 1 author, 2 changes | 0 requests | 0 exceptions
        public IActionResult Index([Bind("Id,D1mm,D2mm,D3mm")] OvaryVis form)
        {
            OvaryVis record = new OvaryVis
            {
                D1mm = form.D1mm,
                D2mm = form.D2mm,
                D3mm = form.D3mm,
                JobSubmitted = DateTime.UtcNow,
                ResultVis = -1,
                StatusMsg = String.Format("Job created at {0}", DateTime.UtcNow.ToString("dd-MM-yy, HH:mm:ss"))
            };
            return RedirectToAction("Result", new { id = record.Id });
        }

        [HttpGet]
        0 references | imageqc, 14 hours ago | 1 author, 2 changes | 0 requests | 0 exceptions
        public IActionResult Result(string Id)
        {
            return View(new OvaryVis());
        }
```

Figure 7 **Home Controller Implementation of Index and Result Actions**

You should note that the views are implemented as Razor views (cshtml), and as such contain various helpers to facilitate integration with the models and controllers. Otherwise they are much like any other HTML form. For example, Index.cshtml with some elements removed, looks much like this:

```
<form id="ScanForm" asp-controller="Home" asp-action="Index" method="post" >
  <div class="form-group">
    <label asp-for="D1mm" class="col-md-2 control-label"></label>
    <input asp-for="D1mm" class="form-control" />
  </div>
    <div class="form-group">
    <input type="submit" value="Submit" class="btn btn-default" />
  </div>
</form>
```

You can copy the files from the source provided with this article, or write the code yourself. Once you're done, you should build the Web App and test it as before. Having implemented the HttpGet Index method in the HomeController and replaced the template HTML in Index.cshtml with the form for populating an OvaryVis object, your Web site will generate a page like the left side of **Figure 8** whenever its URL is entered into a browser's address bar. Adding the HttpPost Index method and the Result.cshtml view means then when you click the Submit button a page like the right side of **Figure 8** appears.



```
Id
JobSubmitted
01/01/0001 00:00:00
D1mm
0
D2mm
0
D3mm
0
StatusMsg
ResultVis
0
```

Figure 8 **The OvaryVisWeb App Input Form and Results view**

Creating the Database

Visual Studio provides the tooling to create a database so your Web App can store its data. By the end of this section, you'll have created a database on the local version of SQL Server installed on your PC with Visual Studio. You'll also use Visual Studio's SQL Server Object Explorer to explore this database.

Creating the database for the model involves very little effort as we are using the code-first approach, which is mostly automated by the Entity Framework (EF) tooling. You need to provide a connection string in appsettings.json with the name of your local SQL server, the name of your database (Msdn.OvaryVisDb) and a few other parameters. For specific details, download the AzureResCmds.txt file provided in the GitHub repository that accompanies this article. The connection string is hooked-up with Entity Framework by adding the following line to the bottom of the ConfigureServices method of the Startup class located in your project's root directory, though you will need also to create the ApplicationDbContext class described below:

```
services.AddDbContext<ApplicationDbContext>(options =>
      options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
```

This way of adding services is called Dependency Injection and means that you can get access to the database context in any controller just by including it as a parameter to its constructor. We'll show how that can be done later, but in the meantime you'll need to add a folder and a class to your project to complete its support for EF. By convention the class is called ApplicationDbContext and it is placed in a folder called Data in the Web App Project directory. This class is implemented as follows:

```
using Microsoft.EntityFrameworkCore;
using OvaryVisWebApp.Models;

public class ApplicationDbContext : DbContext {

  public virtual DbSet<OvaryVis> OvaryVis { get; set; }

  public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
    : base(options)
  { }
}
```

You should note that ApplicationDbContext is derived from DbContext, which is implemented in EntityFrameworkCore, a package that is included in the AspNetCore.App meta package added to your project by the Visual Studio Template for ASP.NET Core 2.1 Web Apps. You are now ready to create your database, but first you need to update the name of your database in the connection string, as given in the Web App's local applications settings file, appsettting.json:

```
"ConnectionStrings": {
  "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=Msdn.OvaryVisDb;
    Trusted_Connection=True;MultipleActiveResultSets=true"
},
```

Now check if your project builds without error (F7), as that is a prerequisite for running the EF Tooling. Once that's done you can open the Package Manager Console (View | Other Windows) and enter the following commands:

```
Add-Migration InitialCreate
Update-Database
```

The first command creates your Migrations folder, as well as the classes EF needs to update your database. The second command applies the migrations to your database, creating it according to your connection string settings as required. After the successful completion of both commands, open the SQL Server Object Explorer from the Visual Studio **View** menu and use its **Add SQL Server** button to connect to your local SQL Server. Check that both the database and the OvaryVis table have been created, as shown in **Figure 9**.

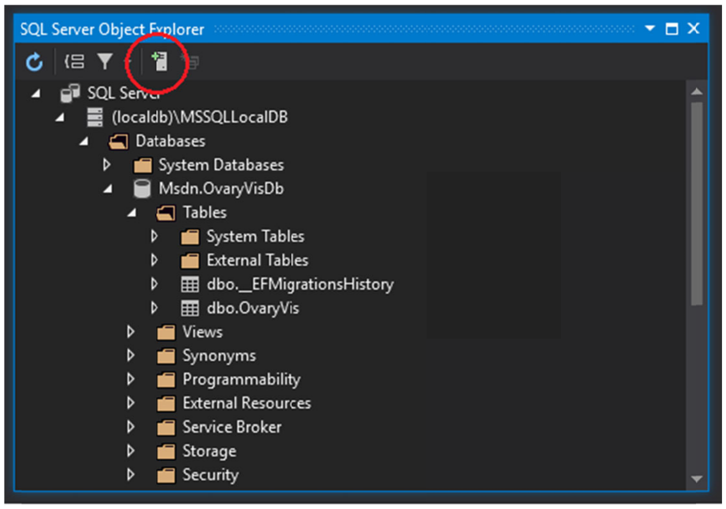


Figure 9 **The Visual Studio SQL Server Object Explorer**

Saving form data in the database is the final step in preparing the OvaryVisWeb App on your local PC. It requires you to add a database context variable to your HomeController, using the dependency injection mechanism mentioned previously. Here's how:

```
using Microsoft.EntityFrameworkCore;
using OvaryVisWebApp.Data;

namespace OvaryVisWebApp.Controllers
{
  public class HomeController : Controller
  {
    private readonly ApplicationDbContext _context;
    public HomeController(ApplicationDbContext context) {_context = context; }
  }
}
```

You then use this context variable to save your OvaryVis object in the Index method just before redirecting to the Result Action (method), as shown in **Figure 10**.

Figure 10 Implementation of HomeController Index Method for HttpPost

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Index([Bind("Id,D1mm,D2mm,D3mm")] OvaryVis form)
{
  OvaryVis record = new OvaryVis
  {
    D1mm = form.D1mm,
    D2mm = form.D2mm,
    D3mm = form.D3mm,
    JobSubmitted = DateTime.UtcNow,
    ResultVis = -1,
    StatusMsg = String.Format("Created {0}",DateTime.UtcNow.ToString("HH:mm:ss"))
  };

  _context.Add(record);
  await _context.SaveChangesAsync();

  return RedirectToAction("Result", new { id = record.Id });
}
```

In this way you can restore the OvaryVis object from the database using its Id value (primary key), as passed from the Index method. You should update both the Index and Result methods to become async Task<> so you can use the more efficient async methods of the context object. A full explanation of using async in C# code is given in Chapters 19 and 20 of Mark Michaelis' book "Essential C#" but basically it's just a mechanism to make better use of the threads created for executing code on your PC's processor.

```
[HttpGet]
public async Task<IActionResult> Result(string Id)
{
  var record = await _context.OvaryVis.SingleOrDefaultAsync(a => a.Id == Id);
  if (record == null)
    record = new OvaryVis();
  return View(record);
}
```

Building and testing your project needs to be performed as before. You should have now created a working Web App on your local PC that collects user input, saves it to a database and then displays the result. You are now ready to publish your Web site on the Azure cloud.

Publishing your Web Site on Azure

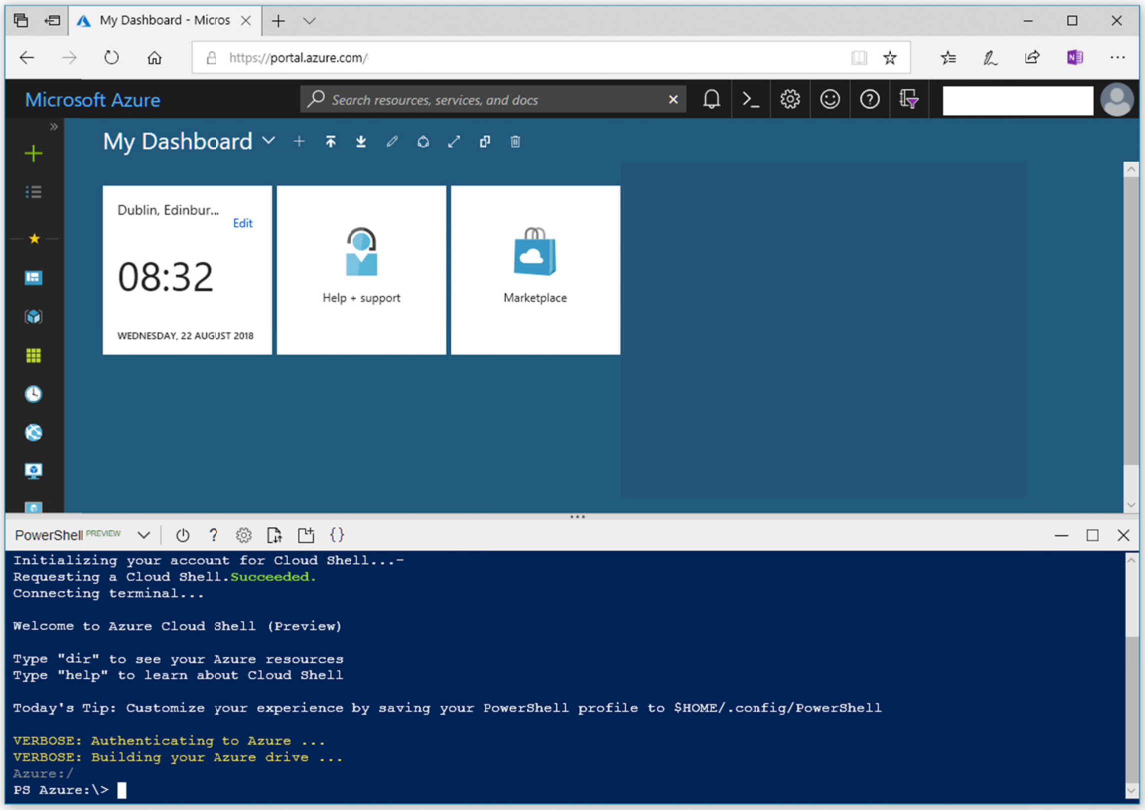
Azure provides a quick, secure, and cost effective way to host a public Web site. You'll need a subscription, but pay-as-you go subscriptions are available to new customers with $200 free credit, which means you can complete this project without incurring any cost. Even if you're an existing customer, the monthly cost of providing the type of Web site described in this article is comparable to buying a couple cups of coffee.

Getting a pay-as-you-go subscription means you pay only for what you use, though as you can't set a credit limit this can be substantial. For this reason, you need to take care when creating resources and keep an eye on your costs by visiting the Billing blade on the Azure Portal at portal.azure.com. If you no longer need your Web site, I recommend that you delete its Resource Group to avoid unexpected costs.

Provisioning Azure Resources for your Web site is something that can be done interactively using the Portal pages, or by issuing commands from the built-in Cloud Shell, or by running scripts using Windows Powershell on your desktop. For the purposes of this article, we recommend issuing commands at the Cloud Shell console, as this is the simplest and quickest way to get your Web site published. However, afterwards you should explore the Portal pages to gain a better understanding about what you've done, and to help manage your Web site going forward.

Opening the Azure Cloud Shell console gives you an easy way to provision your resources from the Azure Portal. You open the console by clicking the '>_' item in the top menu bar that appears when you login. The first time you open the shell you are prompted to select PowerShell or Bash console; I choose PowerShell (shown in **Figure 11**), but Bash works equally well. You are then prompted to create a storage account for your subscription, unless one already exists. The monthly cost of this storage is typically less than a few cents, and it's provisioned just by clicking the 'create storage' button. Just wait a few seconds for the Cloud Shell to initialize.

Once the PowerShell prompt appears, you're ready to start issuing the commands needed to provision the resources for your Web site. These can be found in AzureResCmds.txt located in the Scripts folder of the source code supplied with this article. They can be copied and pasted into the console, but take care to apply the changes detailed below. If you're using the Windows Edge browser, then 'paste' is accessed by right clicking to reveal the context menu.



```
Initializing your account for Cloud Shell...-
Requesting a Cloud Shell.Succeeded.
Connecting terminal...

Welcome to Azure Cloud Shell (Preview)

Type "dir" to see your Azure resources
Type "help" to learn about Cloud Shell

Today's Tip: Customize your experience by saving your PowerShell profile to $HOME/.config/PowerShell

VERBOSE: Authenticating to Azure ...
VERBOSE: Building your Azure drive ...
Azure:/
PS Azure:\>
```

Figure 11 **PowerShell Console Opened in the Azure Cloud Shell**

Creating your Web App requires just the few commands given below, though you'll need to change the subscription name to match that of your own account. You'll also need to replace *MSDNOvaryVisWebApp* with your own unique name as it becomes part of the URL that provides public access to your Web site. In addition, you might want to choose a more appropriate location for hosting your resources than WestEurope.

```
az account set --subscription MsdnOvaryVis
az group create --name resMSDNOvaryVis --location "WestEurope"
az appservice plan create --name MSDNOvaryVisPlan --resource-group resMSDNOvaryVis
  --location "WestEurope" --sku FREE
az webapp create --name MSDNOvaryVisWebApp --plan MSDNOvaryVisPlan
  --resource-group resMSDNOvaryVis
```

Note that –sku FREE is selected, which means that the plan for hosting your Web site costs nothing, but this comes with some limitations that make it suitable only for development and test use, or for the sort of evaluation use intended for a Web site like ours.

Creating your SQL Server and a database is just as easy and requires only the few commands listed below. Again you need to give a unique name for your SQL Server instead of msdnovaryvisdbsvr, and you must change the other commands accordingly. In addition you need to provide appropriate values for Admin-user and Admin-password in place of XXX and YYY and also change AAA to reflect the IP address of your PC (as given in your SQL Server Firewall blade). The only other change to consider is the location name, which should be the same as set for your Web App to avoid the performance hit associated with moving data between different datacenters.

```
az sql server create --name msdnovaryvisdbsvr --resource-group resMSDNOvaryVis
  --location "WestEurope" --admin-user XXX --admin-password YYY
az sql server firewall-rule create --server msdnovaryvisdbsvr
  --resource-group resMSDNOvaryVis --name AllowMyPC --start-ip-address AAA
  --end-ip-address AAA
az sql server firewall-rule create --server msdnovaryvisdbsvr
  --resource-group resMSDNOvaryVis --name AllowAzure --start-ip-address 0.0.0.0
  --end-ip-address 0.0.0.0
az sql db create --server msdnovaryvisdbsvr --resource-group resMSDNOvaryVis
  --name Msdn.OvaryVisDb --service-objective Basic
```

You should be aware that there is a monthly cost of about $5 associated with providing your database, even though the lowest service objective is set at Basic. Higher service objectives cost considerably more, but provide better performance and storage capacity.

Setting your Web App connection string completes the work of setting-up the Azure resources needed for you to publish your Web site. You can obtain the connection string of your SQL Server with the following command, though obviously you need to replace the server name with yours:

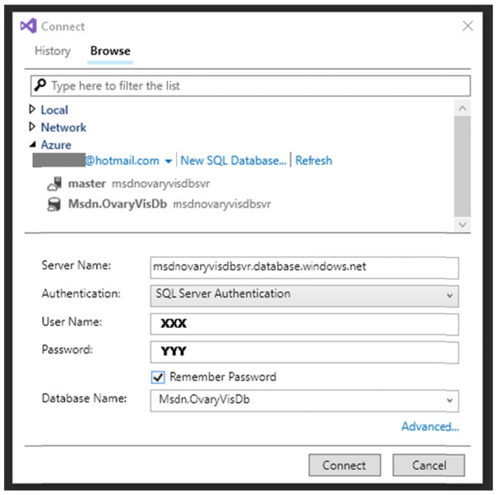
```
az sql db show-connection-string --name Msdn.OvaryVisDb --server msdnovaryvisdbsvr
  --client ado.net --output tsv
```

You may find it convenient to copy the response into Notepad and then change the values for the database server's admin user (XXX) and password **(**YYY) to ones for your own SQL Server. You can then paste this Default Connection value into the following command, remembering of course to change the name of the Web App and its Resource Group to match your own.

```
az webapp config connection-string set --connection-string-type SQLAzure
  --name MSDNOvaryVisWebApp --resource-group resMSDNOvaryVis
  --settings DefaultConnection='Server=tcp:msdnovaryvisdbsvr.database.windows.net,1433;
    Database=Msdn.OvaryVisDb;User ID=XXX;Password=YYY;
    Encrypt=true;Connection Timeout=30;'
```

The successful execution of the command causes your SQL Server connection string to be added to the application settings of your Web App.

Checking access to your Azure SQL Server and its database using SQL Server Object Explorer is a good idea at this point. After opening the Explorer window from Visual Studio, you'll need to add a new connection to your Azure SQL Server. This is achieved like before, but this time instead of connecting to your local SQL Server you want to connect to the one hosted by Azure. To do this, select SQL Server Authentication in the Connect dialog and copy the same server name, user name and password you gave above, as shown in **Figure 12**. After clicking OK the dialog box will close and your SQL Server should appear as the root in the Explorer window, together with the name of your database in its collection of databases. However, if you look at the list tables for your database you'll see that the OvaryVis table is missing. We'll fix that problem next.


Connect

History Browse

Type here to filter the list

Local
Network
Azure
[redacted]@hotmail.com | New SQL Database... | Refresh
master msdnovaryvisdbsvr
Msdn.OvaryVisDb msdnovaryvisdbsvr

Server Name: msdnovaryvisdbsvr.database.windows.net
Authentication: SQL Server Authentication
User Name: XXX
Password: YYY
[✓] Remember Password
Database Name: Msdn.OvaryVisDb

Advanced...

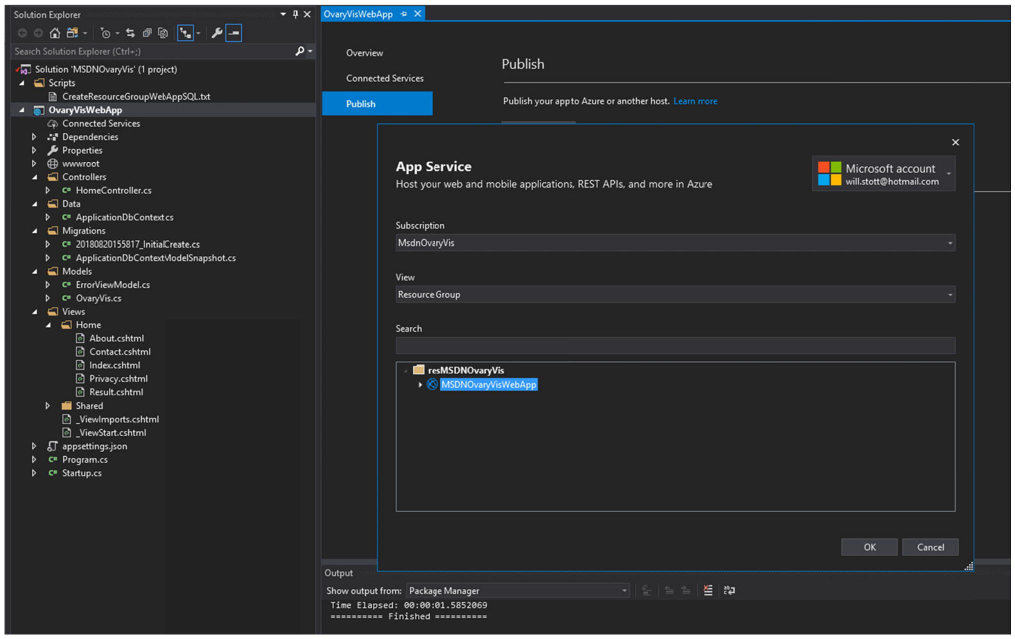
Connect Cancel

Figure 12 **SQL Server Object Explorer Connect Dialog Box**

Applying EF Migrations to your Azure Database will complete the work needed to setup your Azure Resources. The procedure is much the same as when applying EF Migrations to your local database, except that you should change the connection string in appsettings.json to reference your new Azure database. After making this change using Visual Studio, rebuild your project and then apply the update-database command in the Package Manager Console as before. After successfully applying the EF Migrations, you should check that the OvaryVis table now appears in the object explorer, though you will need to perform a refresh or reconnect first. Finally, run the Web App from Visual Studio (Ctrl+F5) and test it as you did previously. This time however the record will be created in your Azure database as the connection is no longer referencing your local database.

Publishing to Azure from Visual Studio is very simple. You just need to select your Web App project in the Solution Explorer window (View | Solution Explorer), right-click and then select Publish to display the Publish window. This allows you to select a publish target. You need to select an existing App Service as you want to publish to the Azure Web App you've just created. After clicking the Publish button you are given a list of services available to your subscription, as shown in **Figure 13**. Once the target is selected and you click OK, publishing happens automatically. Upon completion, your browser opens to the URL of the Web site, which should look the same as when it was opened on your localhost (see **Figure 8)**.

Testing your Web site as deployed to Azure is done in the same way as testing on your local PC, except this time you should use the public URL provided by Azure specifically for your Web App. You can find this URL in the Overview blade of your Web App in Azure Portal. You might also want to check the Application Settings blade and confirm that the database connection string is located in the list of the Web App's connections. When you're satisfied that everything is working, undo the changes you made to the connection string in appsettings.json (select the file, right-click | Undo) so any further development work will use your local database server rather the one you created on the Azure cloud.


Solution 'MSDNOvaryVis' (1 project)
Scripts
CreateResourceGroupWebAppSQL.txt
OvaryVisWebApp
Connected Services
Dependencies
Properties
wwwroot
Controllers
HomeController.cs
Data
ApplicationDbContext.cs
Migrations
20180820155817_InitialCreate.cs
ApplicationDbContextModelSnapshot.cs
Models
ErrorViewModel.cs
OvaryVis.cs
Views
Home
About.cshtml
Contact.cshtml
Index.cshtml
Privacy.cshtml
Result.cshtml
Shared
_ViewImports.cshtml
_ViewStart.cshtml
appsettings.json
Program.cs
Startup.cs

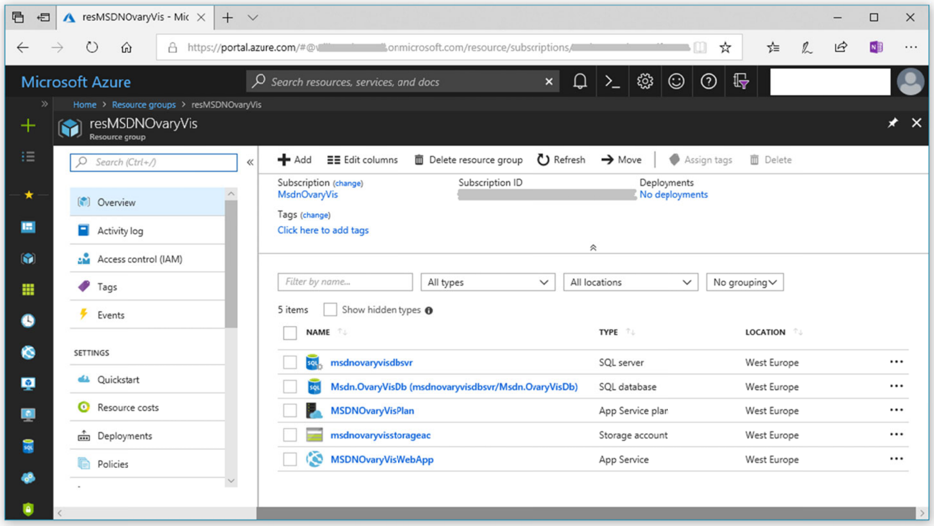
Publish
Publish your app to Azure or another host. Learn more

App Service
Host your web and mobile applications, REST APIs, and more in Azure
Microsoft account
will.stott@hotmail.com
Subscription: MsdnOvaryVis
View: Resource Group
Search:
resMSDNOvaryVis
MSDNOvaryVisWebApp

```
Show output from: Package Manager
Time Elapsed: 00:00:01.5852069
========== Finished ==========
```

Figure 13 **Visual Studio Publishes Your Web App and Associated Files to Azure**

Deleting the resources you've created will avoid any further billing and is achieved by visiting the resource group blade on the Azure Portal and clicking the Delete resource button, as shown in **Figure 14**. Alternatively, you can delete your subscription entirely by visiting its Overview blade, though rest assured you won't get billed unless you're actually using resources. If you want to redeploy your Web site, you'll need to recreate these resources and republish your projects from Visual Studio. However, this should not take you more than a few hours, particularly if you've documenting the commands and use PowerShell. It's good practice always to do this when working with Azure, though it seems likely that before long tools will become available to allow complete automation of your resource deployments together with their associated data.


https://portal.azure.com/#@[redacted]ormicrosoft.com/resource/subscriptions/[redacted]

Home > Resource groups > resMSDNOvaryVis
resMSDNOvaryVis
Resource group

Subscription (change): MsdnOvaryVis
Subscription ID: [redacted]
Deployments: No deployments
Tags (change): Click here to add tags

5 items

NAME	TYPE	LOCATION
msdnovaryvisdbsvr	SQL server	West Europe
Msdn.OvaryVisDb (msdnovaryvisdbsvr/Msdn.OvaryVisDb)	SQL database	West Europe
MSDNOvaryVisPlan	App Service plan	West Europe
msdnovaryvisstorageac	Storage account	West Europe
MSDNOvaryVisWebApp	App Service	West Europe

Figure 14 **Azure Resource Group Containing Resources Needed for the Web site**

Wrapping Up

Hopefully the experience of building and deploying your first Web site to the Azure cloud has been a positive one for you. Let's now build on this foundation to implement a ServiceBus Queue and an Azure Functions so your Web site can handle the sort of long running processes associated with a machine learning service like ovary classification. For more on that, read my article in MSDN Magazine titled, "Web Site Background Processing with Azure Service Bus Queues."

About the author and the reviewer

Dr Will Stott has over 25 years' experience working as contractor/consultant for a wide range of companies in UK and Europe including IBM, Cap-Gemini, Logica CMG and Accenture. However, for the last ten years most of his time has been spent doing research at University College London (UCL) on Ovarian Cancer Screening. Will has spoken at many conferences both in the UK and Internationally. He is also author of papers published in various journals as well as the book "*Visual Studio Team System: Better Software Development for Agile Teams. Addison-Wesley Professional; 2007"*

Mark Brearley is UBS Group Operations Transformation lead for India, but as a manager has no real experience of software development. However, he wanted to know more about Azure and get some hand-on experience of using it. Therefore he followed the article and in a few hours succeeded in building a Web site and database hosted on Azure– see www.markbrearley.com