Agenda: Implement Azure Functions

- About Azure Functions.
- Azure Webjobs vs Azure Functions
- Create an Azure Function
- Create an event processing function
    - Timer Triggered Function
    - Blog Storage Triggered Function
- Implement an Azure-connected functions
    - Adding message to Storage Queue
    - Connecting to SQL Database
- Walkthrough: Creating Image Thumbnail

About Azure Functions

- Azure Functions is a **serverless compute service** that enables you to run code on-demand without having to explicitly provision or manage infrastructure.
- Use Azure Functions to run a script or piece of code in response to a variety of events.
- Functions can make development even more productive, and you can use your development language of choice, such as C#, F#, Java, Node.js, Python or PHP.
- **Consumption Pricing Model**: Pay only for the time your code runs and **trust** Azure to scale as needed.

Functions provides templates to get you started with key scenarios, including the following:

- **HTTPTrigger** - Trigger the execution of your code by using an HTTP request.
- **TimerTrigger** - Execute cleanup or other batch tasks on a predefined schedule.
- **QueueTrigger** - Respond to messages as they arrive in an Azure Storage queue.
- **BlobTrigger** - Process Azure Storage blobs when they are added to containers. You might use this function for image resizing.
- **EventHubTrigger** - Respond to events delivered to an Azure Event Hub. Particularly useful in application instrumentation, user experience or workflow processing, and Internet of Things (IoT) scenarios.
- **Generic Webhook** - Process webhook HTTP requests from any service that supports webhooks.
- **ServiceBusQueueTrigger** - Connect your code to other Azure services or on-premises services by listening to message queues.
- **ServiceBusTopicTrigger** - Connect your code to other Azure services or on-premises services by subscribing to topics.
- …

How much does Functions cost?

Azure Functions has two kinds of pricing plans, choose the one that best fits your needs: +

- **Consumption plan** - When your function runs, Azure provides all of the necessary computational resources. You don't have to worry about resource management, and you only pay for the time that your code runs.
- **App Service plan** - Run your functions just like your web, mobile, and API apps. When you are already using App Service for your other applications, you can run your functions on the same plan at no additional cost.

Functions vs. WebJobs

We can discuss Azure Functions and Azure App Service WebJobs together because they are both code-first integration services and designed for developers. They enable you to run a script or a piece of code in response to various events, such as new Storage Blobs or a WebHook request.

Here are their similarities:

- Both are built on Azure App Service and enjoy features such as source control, authentication, and monitoring.
- Both are developer-focused services.
- Both support standard scripting and programming languages.
- Both have NuGet and NPM support.

Functions is the **natural evolution** of WebJobs in that it takes the best things about WebJobs and improves upon them.

The improvements include:

- Streamlined dev, test, and run of code, directly in the browser.
- Built-in integration with more Azure services and 3rd-party services like GitHub WebHooks.
- Pay-per-use, no need to pay for an App Service plan.
- Automatic, dynamic scaling.
- For existing customers of App Service, running on App Service plan still possible (to take advantage of under-utilized resources).
- Integration with Logic Apps.

The “functionTimeout” in the “**host.json**” file can be configured to timespans ranging from a minimum of 1 second (00:00:01), up to a maximum of 10 minutes (00:10:00). Remember, the default without overriding the setting will be 5 minutes (00:05:00).

Go to FunctionApp → Function app settings → Edit host.json

```
// Set functionTimeout to 10 minutes
{
   "functionTimeout": "00:10:00"
```

```
}
```

Note that overall, this is a big limitation over the alternative of using Azure Web Jobs which do not have any execution runtime timeout limitation.

https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale

One thing worth mentioning is this timeout only applies when running in an consumption plan. When running in a regular app service plan, no time limit applies and your Functions can run as long as you like.

Creating an Azure Function

Create a Function App

1. +Create a resource → Compute → Function App
2. App name = DssDemoFunctions, Hosting Plan: Consumption Plan, Runtime Stack = .NET, Storage Account: <Create New> → Create

Create a Function

3. Function App → Click Functions + → **HTTP trigger**,
4. Name = SayHello,
5. Authorization level = Function

   Authorization level controls whether the function requires an **API key** and which key to use; Function uses a function key; Admin uses your master key. The function and master keys are found in the **'keys'** management panel on the portal, when your function is selected. For user-based authentication, go to Function App Settings.

Following is the function auto generated in C#

```
#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;

public static async Task<IActionResult> Run(HttpRequest req, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    string name = req.Query["name"];
```

```
    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;

    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
}
```

Test the function:

1. In your new function, click **</> Get function URL** and copy the **Function URL.**

   ```
   https://dssdemofunc.azurewebsites.net/api/HttpTriggerJS1?code=HaAojLFnYG05Fa0hMtocj7ymoCExasRWW64BaWtbQGx/O9DvSoDv8A==
   ```

2. Paste the URL for the HTTP request into your browser's address bar. Append the query string `&name=<yourname>` to this URL and execute the request.
3. View the function logs at the bottom of the screen (click up arrow at the bottom of the screen).
4. You can make changes (specially to context.res.body) to the function as needed and test the same.
5. You can also test the function from Right Handside panel: Test (expand to test)

Timer Triggered Function

1. Function App → Click Functions **+** → Choose a template: **TimerTrigger – C#** → Name your function = TimerTriggerCSharp, Schedule= 0 */1 * * * * (CRON expression that schedules your function to run every minute) → Create
2. If required we can update the **timer schedule**: Integrate tab → {seconds} {minutes} {hour} {day} {month} {dayofweek}
   - To trigger once every hour = 0 0 */1 * * *
   - To trigger once every five minutes: 0 */5 * * * *
   - To trigger once at the top of every hour: "0 0 * * * *
   - To trigger once every two hours: 0 0 */2 * * *
   - To trigger once every hour from 9 AM to 5 PM: 0 0 9-17 * * *
   - To trigger At 9:30 AM every day: 0 30 9 * * *
   - To trigger At 9:30 AM every weekday: 0 30 9 * * 1-5

3. When a timer trigger function is invoked, the timer object is passed into the function. The following JSON is an example representation of the timer object.

```
{
  "Schedule":{    },
  "ScheduleStatus": {
    "Last":"2016-10-04T10:15:00.012699+00:00",
    "Next":"2016-10-04T10:20:00+00:00"
  },
  "IsPastDue":false
}
```

Example:

```
using System;
public static void Run(TimerInfo myTimer, TraceWriter log)
{
  if (myTimer.IsPastDue)
  {
    log.Info("Timer is running late!");
  }
  log.Info(myTimer.ScheduleStatus.Last.ToString());
  log.Info($"C# Timer trigger function executed at: {DateTime.Now}");
}
```

Blob Storage Triggered Function

1. Function App → Click Functions **+** → Choose a template: **BlobTrigger** for your desired language. **Name your function**, Path and Storage account connection and then click **Create**.
2. Click **Integrate**,
    a. Blob parameter **name**: The variable name used in function code for the blob.
    b. **Path**: A path that specifies the container to monitor, and optionally a blob name pattern = **"mycontainer/{name}"**

    Examples of Path:

        a. **input/original-{name}**: This path would find a blob named *original-**Blob1**.txt* in the ***input*** container, and the value of the name variable in function code would be Blob1.
        b. ***samples/{name}.png***: only *.png* blobs in the *samples* container will trigger the function.

    c. Expand **Documentation**, and Note **Account name** and **Account key.** These credentials are used to connect to the storage account.

3. Go to Storage Account and Create a container, Name=mycontainer
4. Test the function:
    a. Expand your storage account, **Blob containers**, and **mycontainer**. Click **Upload** and then **Upload files...**.
    b. In the **Upload files** dialog box, click the **Files** field. Browse to a file on your local computer, such as an image file, select it and click **Open** and then **Upload**.
    c. Go back to your function **logs** and verify that the blob has been read.

Add messages to an Azure Storage queue using Functions

In Azure Functions, input and output bindings provide a declarative way to connect to external service data from your function. In this topic, learn how to update an existing function by adding an output binding that sends messages to Azure Queue storage.

Continue the "**Create a HTTP triggered function"** Function

1. Add an output binding: Click Integrate and **+ New output**, → Azure Queue storage → Select
2. Set Message parameter name: outputQueueItem, Queue name=myqueue-items, Storage account connection:<select storage account> → Save
3. **Update the function code → Save**
    - C#:

```
public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, ICollector<string>
outputQueueItem, TraceWriter log)
{
   ....
   outputQueueItem.Add("Name passed to the function: " + name);
   log.Info("System time: " + System.DateTime.Now.ToLongTimeString());
   return name = . . .
}
```

    - JavaScript:

```
context.bindings.outQueueItem = "Name passed to the function: " +
      (req.query.name || req.body.name);
```

Note: Documentation can be explanded to understand various parameters for the Azure Functions

4. Test the function
    a. Run the function and Check the logs to make sure that the function succeeded. A new queue named **outqueue** is created in your Storage account by the Functions runtime when the output binding is first used.
    b. Connect to the storage account queue and verify the new queue and message you added to it.

Connect to SQL Database

1. Create an SQL Database and copy its connection string.
2. Navigate to your function app you created
3. Select **Platform features** > **Application settings.**
4. Scroll down to **Connection strings** and add a connection string "sqldb_connection"
5. Save
6. In your function app, select the timer-triggered function

```
#r "System.Configuration"
#r "System.Data"

using System;
using System.Configuration;
using System.Data.SqlClient;
using System.Threading.Tasks;

public static void Run(TimerInfo myTimer, TraceWriter log)
{
   log.Info($"C# Timer trigger function executed at: {DateTime.Now}");
   var str = ConfigurationManager.ConnectionStrings["sqldb_connection"].ConnectionString;
   using (SqlConnection conn = new SqlConnection(str))
   {
      conn.Open();
      var text = "UPDATE SalesLT.SalesOrderHeader SET [Status] = 5  WHERE ShipDate < GetDate();";
      using (SqlCommand cmd = new SqlCommand(text, conn))
      {
         // Execute the command and log the # rows affected.
         var rows = await cmd.ExecuteNonQueryAsync();
         log.Info($"{rows} rows were updated");
      }
   }
}
```


CRUD Operations using Azure Web App and Background Task using Azure Functions

Following .NET and Azure Features will be covered in this walkthrough

1. ASP.NET MVC Web Application

2. Entity Framework Code First Approach
3. Azure App Service
4. Storage Service
5. SQL Database Service
6. Azure Serverless Functions

Building ASP.NET MVC Web Application with Entity Framework

Create a New ASP.NET MVC Application

1. Visual Studio → File → New → Project, Template = ASP.NET Web Application → MVC, Change Authentication = No Authentication, Check Host in the cloud → OK
2. Add New Project to Solution for sharing code between web application and web jobs project.
    a. File → Add → New Project → Class Library → Name="**CommonDemoLibrary**" → OK
3. Go to Tools → NuGet Package Manager → Manage Nuget Packages for Solution
4. Add reference to **EntityFramework.**Dll in the class library project
5. Add the following in Employee.cs (CommonDemoLibrary Project)

```
public class Employee
{
   [Key]
   public int Id { get; set; }
   public string EmpName { get; set; }
   public decimal Salary { get; set; }
   [StringLength(2083)]
   [DisplayName("Full-size Image")]
   public string ImageURL { get; set; }
   [StringLength(2083)]
   [DisplayName("Thumbnail")]
   public string ThumbnailURL { get; set; }
}
public class MyDemoContext : DbContext
{
   public MyDemoContext() : base("name=MyDemoContext") { }
   public DbSet<Employee> Employees { get; set; }
}
public class BlobInformation
{
   public Uri BlobUri { get; set; }
```

```
  public string BlobName
  {
    get
    {
      return BlobUri.Segments[BlobUri.Segments.Length - 1];
    }
  }
  public string BlobNameWithoutExtension
  {
    get
    {
      return Path.GetFileNameWithoutExtension(BlobName);
    }
  }
  public int EmpId { get; set; }
}
```

6. In Web Application add reference to class library project "**CommonDemoLibrary**"
7. In Web.Config add the following

```
<connectionStrings>
  <add name="MyDemoContext" connectionString="Data Source=.\sqlexpress;Integrated
Security=True;database=DemoDb" providerName="System.Data.SqlClient"/>
 </connectionStrings>
```

8. Build the Solution
9. Web App project → Right Click on Controller Folder → Add → Controller… → Select "MVC 5 Controller with Views, using Entity Framework" → Add
10. Select Model = Employee, DataContext Class = "DemoDbContext" → Add
    Note: This generates Controller along with Views for Add/Edit/Delete operations.
11. In Views/Employees/Create.cshtml and Edit.cshtml replace the HTML for ImageURL and ThumbnailUrl with the following (FileUpload element)

```
<div class="form-group">
        <label class="control-label col-md-2" for="imageFile">Image file</label>
        <div class="col-md-10">
          <input type="file" name="imageFile" accept="image/*" class="form-control fileupload" />
        </div>
</div>
```

12. In Create.cshtml and Edit.cshtml update the BeginForm as below

```
@using (Html.BeginForm("Create", "Employees", FormMethod.Post, new { enctype = "multipart/form-data"
}))
```

13. Run and test the application by inserting Employee. **Note that ImageURL and ThumbNailURL are blank.**

Storing Image as Blob and posting a message in Azure Storage:

14. Azure Portal → Create a New Storage Account
    a. Set Name = "dssemployeestorage" (all lowercase), Set Region or Affinity Group and Replication → Create
    b. Copy the Connection String of the Storage Account
15. Edit the Web.Config, edit <connectionStrings> section set

```
<connectionStrings>
        <add name="MyDemoContext" . . ./>
        <add name="AzureWebJobsDashboard"
connectionString="DefaultEndpointsProtocol=https;AccountName=dssemployeestorage;AccountKey=0z1SuZxDM+
. . ." />
        <add name="AzureWebJobsStorage"
connectionString="DefaultEndpointsProtocol=https;AccountName=dssemployeestorage;AccountKey=0z1SuZxDM+
. . ." />
</connectionStrings>
```

16. Tools → NuGet Package Manager → Manage NuGet Packages for Solution → Search "Windows Azure Storage" → Install.

Note: This add reference to **Microsoft.WindowsAzure.Storage.dll**.

17. Add the following Employees Container

```
public string UploadImage(HttpPostedFileBase imageFile)
{
  //Write code here to Storage Image in Azure Blob Storage and Get the URL of image
  var storageAccount =
CloudStorageAccount.Parse(ConfigurationManager.ConnectionStrings["AzureWebJobsStorage"].ToString());

  //To Upload the Image into the Blob Container
  string blobName = Guid.NewGuid().ToString() + Path.GetExtension(imageFile.FileName);
  var blobClient = storageAccount.CreateCloudBlobClient();
  CloudBlobContainer imagesBlobContainer = blobClient.GetContainerReference("images");
  imagesBlobContainer.CreateIfNotExists();
  CloudBlockBlob imageBlob = imagesBlobContainer.GetBlockBlobReference(blobName);
  var fileStream = imageFile.InputStream;
```

```
    imageBlob.UploadFromStream(fileStream);
    fileStream.Close();
    return imageBlob.Uri.ToString();
}

private void PostMessageToQueue(int empId, string imageUrl)
{
    var storageAccount =
CloudStorageAccount.Parse(ConfigurationManager.ConnectionStrings["AzureWebJobsStorage"].ToString());
    //To create the Queue with BlobInformation
    CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();
    CloudQueue thumbnailRequestQueue = queueClient.GetQueueReference("thumbnailrequest");
    thumbnailRequestQueue.CreateIfNotExists();
    BlobInformation blobInfo = new BlobInformation() { EmpId = empId, BlobUri = new Uri(imageUrl) };
    var queueMessage = new CloudQueueMessage(JsonConvert.SerializeObject(blobInfo));
    thumbnailRequestQueue.AddMessage(queueMessage);
}
```

18. Update EmployeesController → **Create** method to call the above method.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "EmpName,Salary")] Employee employee, HttpPostedFileBase
imageFile)
{
    if (ModelState.IsValid)
    {
        employee.ImageURL = UploadImage(imageFile);
        db.Employees.Add(employee);
        db.SaveChanges();
        //Convert
        PostMessageToQueue(employee.Id, employee.ImageURL);
        return RedirectToAction("Index");
    }
    return View(employee);
}
```

19. **Build and run the application. Note that this time FullSizeImage is updated with BLOB URL and ThumbNailUrl is null.**

Programming Azure Functions

20. File → Add → New Project → Visual C# → Cloud → Azure Functions, Name="**FunctionsDemoApp**"
21. In the dialog box
    a. In Dropdown select "**Azure Function v1 (.NET Framework)**"
    b. Select Queue Trigger, Expand Storage Account → Browse →
        i. In the Dialog, Select the Storage Account Created using Azure Portal → OK
    c. Leave Connection String as blank, Path = thumbnailrequest (Queue Name)
    d. OK
22. Add reference to "CommonDemoLibrary"
23. Add reference to **System.Drawing**
24. Tools → Nuget Package Manager → Manage NuGet Package for Solution… → Add reference to "**EntityFramework**"
25. Tools → Nuget Package Manager → Manage NuGet Package for Solution… → Add reference to "**Storage Account**"
26. Edit **local.settings.json** as below

```
{
 "IsEncrypted": false,
 "Values": {
  "AzureWebJobsStorage":
"DefaultEndpointsProtocol=https;AccountName=dssdemostorage;AccountKey=2mszgzSAkAu/ACJKpFQWadiNLbvX
PexYUFo6CwMkg7X3N06I3SdIevdLBkJhEOg17p+mbignhq/7GkybbIKCqA==;BlobEndpoint=https://dssdemostorage.
blob.core.windows.net/;TableEndpoint=https://dssdemostorage.table.core.windows.net/;QueueEndpoint=https:/
/dssdemostorage.queue.core.windows.net/;FileEndpoint=https://dssdemostorage.file.core.windows.net/",
  "AzureWebJobsDashboard":
"DefaultEndpointsProtocol=https;AccountName=dssdemostorage;AccountKey=2mszgzSAkAu/ACJKpFQWadiNLbvX
PexYUFo6CwMkg7X3N06I3SdIevdLBkJhEOg17p+mbignhq/7GkybbIKCqA==;BlobEndpoint=https://dssdemostorage.
blob.core.windows.net/;TableEndpoint=https://dssdemostorage.table.core.windows.net/;QueueEndpoint=https:/
/dssdemostorage.queue.core.windows.net/;FileEndpoint=https://dssdemostorage.file.core.windows.net/"
 },
 "connectionStrings": {
  "MyDemoContext": {
   "ConnectionString": "Server=tcp:dssdemoserver.database.windows.net,1433;Initial Catalog=dssdemodb;Persist
Security Info=False;User
```

```
ID=dssadmin;Password=Password@123;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False
;Connection Timeout=30;",
    "ProviderName": "System.Data.SqlClient"
  }
 }
}
```

27. Edit **Function1.cs** as below

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.IO;
using Microsoft.Azure.WebJobs;
using Microsoft.Extensions.Logging;
using Microsoft.WindowsAzure.Storage.Blob;
using MySharedLibrary;

namespace FunctionsDemoApp
{
  public static class Function1
  {
    [FunctionName("Function1")]
    public static void GenerateThumbnail(
      [QueueTrigger("thumbnailrequest")] BlobInformation blobInfo,
      [Blob("images/{BlobName}", FileAccess.Read)] Stream input,
      [Blob("images/{BlobNameWithoutExtension}_thumbnail.jpg")] CloudBlockBlob outputBlob, ILogger log)
    {
      using (Stream output = outputBlob.OpenWrite())
      {
        ConvertImageToThumbnailJPG(input, output);
        outputBlob.Properties.ContentType = "image/jpeg";
      }

      //SqlConnection connection = new SqlConnection
      //{
      //   ConnectionString = Environment.GetEnvironmentVariable("MyDemoContext")
      //};
```

```
        //string sql = $"Update employees Set ThumbnailUrl = '{outputBlob.Uri.ToString()}' where
Id={blobInfo.EmpId}";
        //SqlCommand cmd = new SqlCommand(sql,connection);
        //connection.Open();
        //cmd.ExecuteNonQuery();
        //connection.Close();
        //log.LogInformation("Connection Closed");
        using (MyDemoContext db = new MyDemoContext())
        {
            var id = blobInfo.EmpId;
            Employee emp = db.Employees.Find(id);
            if (emp == null)
            {
                throw new Exception(String.Format("EmpId: {0} not found, can't create thumbnail", id.ToString()));
            }
            emp.ThumbnailURL = outputBlob.Uri.ToString();
            db.SaveChanges();
        }
    }

    public static void ConvertImageToThumbnailJPG(Stream input, Stream output)
    {
        int thumbnailsize = 80;
        int width;
        int height;
        var originalImage = new Bitmap(input);

        if (originalImage.Width > originalImage.Height)
        {
            width = thumbnailsize;
            height = thumbnailsize * originalImage.Height / originalImage.Width;
        }
        else
        {
            height = thumbnailsize;
            width = thumbnailsize * originalImage.Width / originalImage.Height;
```

```
        }

        Bitmap thumbnailImage = null;
        try
        {
            thumbnailImage = new Bitmap(width, height);

            using (Graphics graphics = Graphics.FromImage(thumbnailImage))
            {
                graphics.InterpolationMode = InterpolationMode.HighQualityBicubic;
                graphics.SmoothingMode = SmoothingMode.AntiAlias;
                graphics.PixelOffsetMode = PixelOffsetMode.HighQuality;
                graphics.DrawImage(originalImage, 0, 0, width, height);
            }
            thumbnailImage.Save(output, System.Drawing.Imaging.ImageFormat.Jpeg);
        }
        finally
        {
            if (thumbnailImage != null)
            {
                thumbnailImage.Dispose();
            }
        }
    }
  }
}
```

28. Test the Application Locally
    a. Run the Azure Function Project
    b. Run the WebAPplication Project
        i. Add a New Employee
    c. Note that the Azure Function execute GenerateThumbnail as the message was posted into the queue
    d. Refresh the Employee list in web application and note that Thumbnail URL is available.
29. Publish the Web Application and Azure Function to Azure Portal