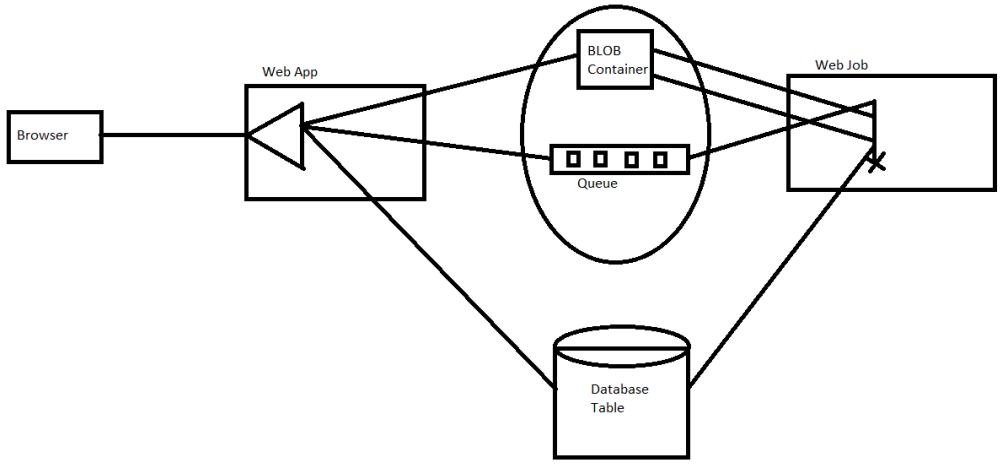
Agenda: Azure Web Jobs

- Introduction
- Developing and Hosting Web Jobs Application
- Azure WebJobs SDK
- Sample Application including WebApps and WebJobs.

Azure Web Jobs



- Azure Web Apps has the ability to run **background processes** to augment your web application. This feature is called Azure WebJobs.
- Azure WebJobs addresses a need common to many websites, which is to offload time-consuming or CPU-intensive tasks to another process **for doing regular jobs and batch work in the background**.
- The **Azure WebJobs Dashboard** provides a nice web interface where you can view, monitor, and invoke your web jobs.

Here's some typical scenarios that would be great for the Windows Azure WebJobs SDK:

- Image processing or other CPU-intensive work.
- Queue processing.
- RSS aggregation.
- File maintenance, such as aggregating or cleaning up log files.
- Other long-running tasks that you want to run in a background thread, such as sending emails.

The following are all valid candidates for implementing a web job:

- .cmd,
- .bat,
- .exe (Windows)
- .ps1 (PowerShell)
- .sh (Bash)
- .php

- .py (Python)
- .js (Node.Js)

Developing, Deploying and Monitoring Web Jobs using Visual Studio

1. Open Visual Studio.Net as Administrator
2. Create a Console Application with Main method as below:

```
class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("Hello From Web Job");
   }
}
```

3. Right click on Project → Publish as Azure Web Job
4. This will open a dialog where you can specify the name of the web job and configure how you want it to run (WebJob run mode = On-Demand).
   a. **Continuous**: A continuous web job is always running and therefore may be implemented using a looping structure such as a while loop.
      Note: WebApp should have AlwaysOn="True" (Property is found under Portal → App Service → Web App → Applicaiton Settings)
   b. **On-Demand**: Web job will be invoked only when you specifically take action to invoke it. You can invoke the web job by using the Azure portal, programmatically by using the WebJobs REST API, or by using Windows PowerShell
5. In the Publish Web dialog, click Microsoft Azure Web Apps → Provide inputs → Publish

Note:

- It's published in the folder: **<WebAppRoot>\app_data\jobs\triggered\<WebJobName>**
- **webjob-publish-settings.json file will be added to the project and can be modified if needed.**
- WebJobs runs on the same machine hardware that the Azure web app is running on but in a different process. (Ensure that Azure web app → Application Settings → Always on = On)

6. **To Invoke the WebJob manually**:
   Visual Studio → Server Explorer → Azure → App Service → <Resource Group> → <App Service Name> → Web Jobs → On Demand & Scheduled → <WebJobName> → Right Click → **Run**
7. **View the WebJobs Dashboard:**
   Visual Studio → Server Explorer → Azure → App Service → <Resource Group> → <App Service Name> → Web Jobs → On Demand & Scheduled → <WebJobName> → Right Click → **View Dashboard**

8. To Create a WebJob From Azure Portal:

1. App Services → <Web App Name> → Settings → Web Jobs → Add (In Blade)
2. Set Name, How to Run and File Upload values

Note: If your web job is a script or .exe that has dependencies on other files such as scripts, DLLs, or configuration files, you must zip the files and upload the .zip file. When you upload a .zip file, the web app environment will automatically unzip the contents in a file location designated for web jobs.

Introduction to the Azure WebJobs SDK

- The purpose of the WebJobs SDK is to simplify the code you write for common tasks that a WebJob can perform, such as image processing, queue processing, RSS aggregation, file maintenance, and sending emails.
- The WebJobs SDK has built-in features for working with Azure Storage and Service Bus, for scheduling tasks and handling errors, and for many other common scenarios.

Benefits of SDK:

1. The WebJobs SDK framework knows when to call your methods and what parameter values to use based on the WebJobs SDK attributes you use in them.
2. The SDK provides ***triggers*** that specify what conditions cause the function to be called, and ***binders*** that specify how to get information into and out of method parameters.
3. The **QueueTrigger** attribute causes a function to be called when a message is received on a queue, and if the message format is JSON for a byte array or a custom type, the message is automatically deserialized.
4. The **BlobTrigger** attribute triggers a process whenever a new blob is created in an Azure Storage account.

```
public static void Main()
{
    JobHost host = new JobHost();
    host.RunAndBlock();
}

public static void ProcessQueueMessage([QueueTrigger("webjobsqueue")] string inputText,
    [Blob("containername/blobname")]TextWriter writer)
{
    writer.WriteLine(inputText); //Writes to the Blob - containername/blobname
}
```

The **JobHost** object is a container for a set of background functions. The **JobHost** object monitors the functions, watches for events that trigger them, and executes the functions when trigger events occur. You call a JobHost

method to indicate whether you want the container process to run on the current thread or a background thread. In the example, the RunAndBlock method runs the process continuously on the current thread.

Using Azure queue storage with the WebJobs SDK:

Step 1: Create a Storage Account

a. Visual Studio → Server Explorer → Connect to Azure → Storage → Right Click and Select **Create Storage Account**

b. Set Subscription, Name="dsswebjobsdemostorage", Region = Southeast Asia, Replication=Locally Redundant → Create

Step 2: Visual Studio New Project

2. Visual C# → Console Application
3. Add the reference to the Nuget Package **Microsoft.Azure.WebJobs**
4. Visual Studio → Server Explorer → Storage → Right Click on dsswebjobsdemostorage → Properties → In properties window → Click on … for Connection Strings property → from the dialog copy Primary Key
5. Edit the App.Config, edit <connectionStrings> section set proper value for NAME and KEY placeholders

```
<connectionStrings>
        <add name="AzureWebJobsDashboard" connectionString="<copy from portal>" />
        <add name="AzureWebJobsStorage" connectionString="<copy from portal>" />
</connectionStrings>
```

6. Edit Functions.cs, comment all the existing code and add the following

```
public static void ProcessStringMessage([QueueTrigger("stringqueue")] string msg, TextWriter logger)
{
   Console.WriteLine(msg); //Writes the msg to the Log of WebJobs, Dashboard can be used to view.
}
public class Employee
{
   public int Id { get; set; }
   public string Name { get; set; }
   public decimal Salary { get; set; }
}
public static void ProcessEmployeeMessage([QueueTrigger("employeequeue")] Employee emp, TextWriter logger)
{
   Console.WriteLine(emp.Id + " " + emp.Name + " " + emp.Salary);
}
```

7. **Run the application (Ctrl + F5).**

Step 3: Writing client application to post messages in Queue:

8. Create a New Console Based Application

9. Add the following to App.Config

```
<appSettings>
 <add key="AzureWebJobsStorage"
value="DefaultEndpointsProtocol=https;AccountName=webjobsstoratedemo;AccountKey=A3/9XagWPC6yoyIgprk
PUxBagbI5yx7oreF+CTxb5pMWujtAi0m553yAc77SIj1f4OZMmIVBgy0ud0bBeGlHQQ==" />
</appSettings>
```

10. Edit Program.cs

```
public class Employee
{
   public int Id { get; set; }
   public string Name { get; set; }
   public decimal Salary { get; set; }
}
class Program
{
   static void Main(string[] args)
   {
      string webJobsStorage = ConfigurationManager.AppSettings["AzureWebJobsStorage"];
      CloudStorageAccount storageAccount = CloudStorageAccount.Parse(webJobsStorage);
      CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

      CloudQueue strQueue = queueClient.GetQueueReference("stringqueue");
      strQueue.CreateIfNotExists();
      string msg = "Hello From the Client Application";
      strQueue.AddMessage(new CloudQueueMessage(msg));

      CloudQueue empQueue = queueClient.GetQueueReference("employeequeue");
      empQueue.CreateIfNotExists();
      Employee emp = new Employee() { Id = 1, Name = "E1", Salary = 10000 };
      empQueue.AddMessage(new CloudQueueMessage(JsonConvert.SerializeObject(emp)));
   }
}
```

11. Run the Application (Ctrl + F5).

Note that the client application has posted the message into the two queues "stringqueue" and "employeequeue" and the same were automatically processed by WebJobs application.

To get queue or queue message metadata:

```
public static void ProcessStringMessage([QueueTrigger("stringqueue")] string msg,
                                        DateTimeOffset expirationTime,
                                        DateTimeOffset insertionTime,
                                        DateTimeOffset nextVisibleTime,
                                        string id,
                                        string popReceipt,
                                        int dequeueCount,
                                        string queueTrigger,
                                        CloudStorageAccount cloudStorageAccount,
                                        TextWriter logger)
{
   Console.WriteLine(
      "logMessage={0}\n" +
      "expirationTime={1}\ninsertionTime={2}\n" +
      "nextVisibleTime={3}\n" +
      "id={4}\npopReceipt={5}\ndequeueCount={6}\n" +
      "queue endpoint={7} queueTrigger={8}",
      msg, expirationTime,
      insertionTime,
      nextVisibleTime, id,
      popReceipt, dequeueCount,
      cloudStorageAccount.QueueEndpoint,
      queueTrigger);
}
```

How to create a queue message while processing a queue message

```
public static void TransformStringToEmployee([QueueTrigger("stringqueue")] string queueMessage,
[Queue("employeequeue")] out Employee outputQueueMessage)
{
```

```
    var ar = queueMessage.Split(',');
    outputQueueMessage = new Employee() { Id = int.Parse(ar[0]), Name = ar[1], Salary = decimal.Parse(ar[2]) };
}
```

Blob Operations:

Following example create a blob with Id of employee and writes to it the properties of employee object added to the queue

```
public static void ProcessEmployeeMessage([QueueTrigger("employeequeue")] Employee emp,
[Blob("employees/{Id}")] TextWriter writer)
{
    Console.WriteLine(emp.Id + " " + emp.Name + " " + emp.Salary);
    writer.WriteLine(emp.Id + " " + emp.Name + " " + emp.Salary);
}
```

Following example create a blob with Id of employee and writes to it the string

```
public static void ProcessEmployee([QueueTrigger("employeequeue")] Employee emp, [Blob("employees/{Id}")]
out string empBlob)
{
    empBlob = "An employee with Name: " + emp.Name + " is added to the queue" ;
}
```

String queue messages triggering blob operations: The following example uses Stream objects to read and write blobs. The queue message is the name of a blob located in the textblobs container. A copy of the blob with "-new" appended to the name is created in the same container.

```
public static void ProcessQueueMessage(
  [QueueTrigger("blobcopyqueue")] string blobName,
  [Blob("textblobs/{queueTrigger}", FileAccess.Read)] Stream blobInput,
  [Blob("textblobs/{queueTrigger}-new", FileAccess.Write)] Stream blobOutput)
{
    blobInput.CopyTo(blobOutput, 4096);
}
```

POCO (Plain Old CLR Object) queue messages: The following example copies a blob to a new blob with a different name. The queue message is a Employee object that includes EmpId and EmpName properties. The property names are used as placeholders in the blob path for the Blob attributes.

```
public static void CopyBlobPOCO(
  [QueueTrigger("copyblobqueue")] Employee blobInfo,
  [Blob("textblobs/{EmpId}", FileAccess.Read)] Stream blobInput,
  [Blob("textblobs/{EmpName}.txt", FileAccess.Write)] Stream blobOutput)
{
   blobInput.CopyTo(blobOutput, 4096);
}
```

Use WebJobs SDK attributes in the body of a function

If you need to do some work in your function before using a WebJobs SDK attribute such as Queue, Blob, or Table, you can use the **IBinder** interface.

The following example takes an input queue message and creates a new message with the same content in an output queue. The output queue name is set by code in the body of the function.

```
public static void CreateQueueMessage([QueueTrigger("inputqueue")] string queueMessage, IBinder binder)
{
   string outputQueueName = "outputqueue-" + DateTime.Now.Month.ToString();
   QueueAttribute queueAttribute = new QueueAttribute(outputQueueName);
   CloudQueue outputQueue = binder.Bind<CloudQueue>(queueAttribute);
   outputQueue.AddMessage(new CloudQueueMessage(queueMessage));
}
```

The following example reads a message from the "employeequeue" queue and writes a blob in the "employees" container. Name of the blob is Id of employee

```
public static void QueueToBlob([QueueTrigger("employeequeue")] Employee emp, IBinder binder)
{
   TextWriter writer = binder.Bind<TextWriter>(new BlobAttribute(DateTime.Now.Month.ToString() + "/" + emp.Id
));
   writer.Write("Completed");
}
```

How to handle poison messages

The SDK will call a function up to 5 times to process a queue message. If the **fifth** try fails, the message is moved to a poison queue.

The poison queue is named ***{originalqueuename}*-poison**. You can write a function to process messages from the poison queue by logging them or sending a notification that manual attention is needed.

In the following example

1. The CopyBlob function will fail when a queue message contains the name of a blob that doesn't exist. When that happens, the message is moved from the copyblobqueue queue to the copyblobqueue-poison queue.
2. The ProcessPoisonMessage then logs the poison message.

```
public static void CopyBlob( [QueueTrigger("copyblobqueue")] string blobName,
    [Blob("textblobs/{queueTrigger}", FileAccess.Read)] Stream blobInput,
    [Blob("textblobs/{queueTrigger}-new", FileAccess.Write)] Stream blobOutput)
{
  blobInput.CopyTo(blobOutput, 4096);
}
public static void ProcessPoisonMessage( [QueueTrigger("copyblobqueue-poison")] string blobName, TextWriter
logger)
{
  logger.WriteLine("Failed to copy blob, name=" + blobName);
}
```

How to trigger a function manually

To trigger a function manually, use the Call or CallAsync method on the JobHost object and the NoAutomaticTrigger attribute on the function, as shown in the following example.

```
public class Program
{
  static void Main(string[] args)
  {
    JobHost host = new JobHost();
    host.Call(typeof(Program).GetMethod("CreateQueueMessage"), new { value = "Hello world!" });
  }

  [NoAutomaticTrigger]
  public static void CreateQueueMessage(
    TextWriter logger,
    string value,
    [Queue("outputqueue")] out string message)
  {
    message = value;
    logger.WriteLine("Creating queue message: ", message);
```

```
    }
}
```

Reference Material:

https://azure.microsoft.com/en-in/documentation/articles/websites-dotnet-webjobs-sdk-storage-queues-how-to/

How to trigger a function when a blob is created or updated

Note: The WebJobs SDK scans log files to watch for new or changed blobs. This process is inherently slow; a function might not get triggered until several minutes or longer after the blob is created. If your application needs to process blobs immediately, the recommended method is to **create a queue** message when you create the blob, and use the QueueTrigger attribute instead of the BlobTrigger attribute on the function that processes the blob.

Single placeholder for blob name with extension:

This code copies only blobs that have names beginning with "original-". For example, *original-Blob1.txt* in the *input* container is copied to *copy-Blob1.txt* in the *output* container.

```
public static void CopyBlob([BlobTrigger("input/original-{name}")] TextReader input, [Blob("output/copy-{name}")]
out string output)
{
    output = input.ReadToEnd();
}
```

Note: You need not specify a name pattern with the blob name placeholder

Separate blob name and extension placeholders:

The following code sample changes the file extension as it copies blobs that appear in the *input* container to the *output* container. The code logs the extension of the *input* blob and sets the extension of the *output* blob to *.txt*.

```
public static void CopyBlobToTxtFile([BlobTrigger("input/{name}.{ext}")] TextReader input,
                                      [Blob("output/{name}.txt")] out string output,
                                      string name,
                                      string ext,
                                      TextWriter logger)
{
    logger.WriteLine("Blob name:" + name);
    logger.WriteLine("Blob extension:" + ext);
    output = input.ReadToEnd();
}
```

How to handle poison blobs:

When a BlobTrigger function fails, the SDK calls it again, in case the failure was caused by a transient error. If the failure is caused by the content of the blob, the function fails every time it tries to process the blob. By default, the SDK calls a function up to 5 times for a given blob. If the fifth try fails, the SDK adds a message to a queue named ***webjobs-blobtrigger-poison*.**

```
public class PoisonBlobMessage
{
   public string FunctionId { get; set; }
   public string BlobType { get; set; }
   public string ContainerName { get; set; }
   public string BlobName { get; set; }
   public string ETag { get; set; }
}
public static void CopyBlob([BlobTrigger("input/{name}")] TextReader input, [Blob("textblobs/output-{name}")] out
string output)
 {
    throw new Exception("Exception for testing poison blob handling");
    output = input.ReadToEnd();
 }

 public static void LogPoisonBlob([QueueTrigger("webjobs-blobtrigger-poison")] PoisonBlobMessage message,
TextWriter logger)
 {
    logger.WriteLine("FunctionId: {0}", message.FunctionId);
    logger.WriteLine("BlobType: {0}", message.BlobType);
    logger.WriteLine("ContainerName: {0}", message.ContainerName);
    logger.WriteLine("BlobName: {0}", message.BlobName);
    logger.WriteLine("ETag: {0}", message.ETag);
 }
```

Example for Creating Image Thumbnail Through Web Jobs

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 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. Create Storage Account using Azure Portal.

    Set Name = "dssemployeestorage" (all lowercase)
15. Edit the Web.Config, edit <connectionStrings> section set

```
<connectionStrings>
```

```
        <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.
17. Add the following **EmployeesController** Class

```
private Uri UploadAndSaveBlobAndPostMessageToQueue(HttpPostedFileBase imageFile, int empId)
{
   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();

   //To create the Queue with BlobInformation
   CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();
   CloudQueue thumbnailRequestQueue = queueClient.GetQueueReference("thumbnailrequest");
   thumbnailRequestQueue.CreateIfNotExists();
   BlobInformation blobInfo = new BlobInformation() { EmpId = empId, BlobUri = imageBlob.Uri };
   var queueMessage = new CloudQueueMessage(JsonConvert.SerializeObject(blobInfo));
   thumbnailRequestQueue.AddMessage(queueMessage);
   return imageBlob.Uri;
}
```

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)
  {
    db.Employees.Add(employee);
    db.SaveChanges();
    if (imageFile != null && imageFile.ContentLength != 0)
    {
      employee.ImageURL = UploadAndSaveBlobAndPostMessageToQueue(imageFile, employee.Id).ToString();
      db.SaveChanges(); //Save the URL of Image
    }
    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 Web Jobs

20. File → Add → New Project → Visual C# → Web → **Cloud** → **Azure WebJobs** (.NET Framework) Name="**WebJobsDemoApp**"
21. Add reference to "CommonDemoLibrary"
22. Add reference to **System.Drawing** and **System.Configuration**
23. Tools → Nuget Package Manager → Manage NuGet Package for Solution… → Add reference to "**EntityFramework**"
24. Copy <**connectionStrings**> section from web.config of Web project into App.Config of WebJob Application
25. Add code as below for **Program** and **Functions** classes

```
class Program
{
    static void Main(string[] args)
    {
      JobHost host = new JobHost();
```

```
        host.RunAndBlock();
    }
}

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

        // Entity Framework context class is not thread-safe, so it must
        // be instantiated and disposed within the function.
        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, ImageFormat.Jpeg);
}
finally
{
   if (thumbnailImage != null)
   {
      thumbnailImage.Dispose();
```

```
        }
      }
    }
}
```

26. Run the Web Application and add a New Employee, note that we only get ImageUrl but not ThumbnailUrl
27. Run the Web Jobs application, Queue will be processed and updates the ThumbnailUrl of the corresponding employee.
28. Refresh the page in web browser and note that even ThumbnailUrl is now updated.

Optional steps to upload the applications to live domain:

29. Deploy the Web Application as Azure Web App
30. Upload the Web Jobs EXE file as explained in earlier examples. Choose mode as "Continuous Job"
31. Also the Web App should have Always Run = True (Application Settings)