

CS526 Enterprise and Cloud Computing
Stevens Institute of Technology—Fall 2022
Assignment Five—Serverless Computing

This assignment requires the use of Visual Studio 2022 or Visual Studio for Mac. You are provided with two ASP.NET projects (Version 7.0, C#):

1. A Web application similar to that for the previous assignment, but with some changes in its backend processing.
2. A project that contains three functions to be deployed in Azure Functions.

In the previous assignment, operations were performed in a synchronous fashion. For example, uploading an image consisted of adding a metadata record to SQL Database, and then uploading the image itself to blob storage. The Web application does not respond back to user until all of these steps are completed. Similarly, when an administrator approves an image, the application does not respond until the database record has been updated.

In this assignment, we introduce intermediate queues for asynchronous processing. When an image is submitted for upload, the application responds immediately back to the user while the image is uploaded to blob storage. No attempt is made at this point to add a metadata record to the database. Instead, the metadata is included with the image upload. Completion of the upload triggers an Azure function, `UploadResponder`, that inserts the metadata into a message and adds this message to an Azure storage queue called `approval-requests`.

When an image approver reviews the images awaiting approval, the metadata for these images is taken from the `approvals request` queue. If an image is approved, its metadata is added to another queue, `approved-images`. Insertion of a message into this queue triggers an Azure function, `ApproveResponder`, that inserts the metadata for the image into the database. If an image is not approved, its metadata is added to another queue, `rejected-images`. Insertion of a message into this queue triggers an Azure function, `RejectResponder`, that deletes the image from blob storage.

For testing purposes, you will want to use the Consumption Plan for Azure Functions. However, this does not support the use of SQL Database in a function. So, although we continue to use SQL Database to store user information for authentication and authorization, we store the image metadata in Table storage¹. The partition key for an image's metadata record is the primary key for the user that uploaded that image, so that metadata for all images for a user are stored in the same partition. This means that the URI for an image is of the form:

`https://.../Images/action-name/user-key/image-key`

¹ We could use Cosmos DB to store both user data and image metadata. We use Table storage for simplicity, because it is supported for local development in the Azurite storage emulator.

The routing logic for this is specified in the application builder:

```
app.MapControllerRoute(  
    name: "image",  
    pattern: "Images/{action}/{userId}/{id}",  
    defaults: new { controller = "Images" });
```

This is then used in tag helpers to specify links to actions for images:

```
<a asp-action="Details" asp-route-userId="@image.getUserId()"  
    asp-route-id="@image.getId()">Details</a>
```

`getUserId()` and `getId()` are helper functions in an image entity object, returning its partition key and row key, respectively.

In the previous assignment, the Web application waited for an image to finish uploading before providing a response to the user:

```
await blobClient.UploadAsync(...);
```

In this assignment, in the spirit of serverless computing, the application may respond to the user immediately, while the upload proceeds in the background, with any exceptions in background processing logged on that thread:

```
blobClient.UploadAsync(...)  
    .ContinueWith(t => logger.LogError(t.Exception.Message),  
        TaskContinuationOptions.OnlyOnFaulted);  
return Task.CompletedTask;
```

The Azure functions are annotated to specify their triggers, and their queue outputs where appropriate. For example, the function that responds to uploads to blob storage is given by:

```
[Function("UploadResponder")]  
[QueueOutput("approval-requests")]  
public string Run(  
    [BlobTrigger("images/{blobname}",  
        Connection = "StorageConnectionString")] string myBlob,  
    string blobname,  
    BlobProperties blobProperties,  
    IDictionary<string, string> metadata,  
    FunctionContext _context)
```

The connection string is specified as the value of the `StorageConnectionString` property in `local.settings.json`. This descriptor is only used for development and is not included in a deployment in the cloud, so the connection string must be specified as the value of the `StorageConnectionString` environment variable when the

function is deployed. When this function returns a string, it is inserted as the payload of a message in the specified queue. Azure queues require that the payload be an ASCII string representing a Base64 encoding:

```
string json = JsonConvert.SerializeObject(image);  
byte[] data = Encoding.UTF8.GetBytes(json);  
return Convert.ToBase64String(data);
```

For development purposes, you should use the Azurite² local Azure storage emulator to test usage of the Blob and Table storage locally. You can use SQL Server as in previous assignments as a local database server while development. The descriptor `launchSettings.json` specifies profiles for running the Web app in Development mode and in Production mode. Use the latter when running in the cloud. You will need to edit the descriptor `appsettings.json` files with your production connection strings, after setting up storage in Azure. The development descriptor `appsettings.Development.json` specifies connection strings for the Azurite storage emulator. You can use the Azure Storage Explorer³ to view the storage, either in Azurite or in Azure.

When developing locally, you can only run one app at a time in Visual Studio. Therefore you need to run the Web app to upload an image to blob storage, then run the functions project to process the upload and generate an approval request, then run the Web app again to grant approval, then run the functions project again to process the approval and insert the image metadata in the images table, then run the Web app again to confirm that the image can now be seen via the Web app. You will need to run a similar sequence to reject an image, confirming that it has been deleted from blob storage.

Once you have your application running, deploy your application in Azure App Service, and your functions in Azure Functions. You should be able to view Azure storage using Storage Explorer as processing proceeds. You should demonstrate a scenario when no images are awaiting approval, then upload an image, then see that there is now a message in the approval-requests queue, then see that image awaiting approval in the Web app, then once this is approved there should be a record in the images table (after processing a message inserted into the approved-images queue by the Web app once approval is granted), and you can view this image in the list of all images in the Web app.

Submission:

Submit your assignment as a zip archive file. This archive file should contain a single folder with your name, with an underscore between first and last name. For example, if your name is Humphrey Bogart, the folder should be named

² <https://learn.microsoft.com/en-us/azure/storage/common/storage-use-azurite?tabs=visual-studio>.

³ <https://azure.microsoft.com/en-us/products/storage/storage-explorer/>

Humphrey_Bogart. This folder should contain a single folder for your solution named ImageSharingServerless, with projects ImageSharingWithServerless, ImageSharingFunctions and ImageSharingModels (providing some definitions used by both the Web app and the Azure functions).

In addition, record mpeg videos demonstrating your deployment working. See the rubric for further details of what is expected. Make sure that your name appears at the beginning of the video, for example as the name of the administration user who manages the Web app. *Do not provide private information such as your email or cwid in the video.* Be careful of any “free” apps that you download to do the recording, there are known cases of such apps containing Trojan horses, including key loggers.