CS526 Enterprise and Cloud Computing
Stevens Institute of Technology—Fall 2024
Assignment Four—Serverless Computing

This assignment requires the use of JetBrains Rider and Azure. You are provided with two ASP.NET projects (Version 9.0[1], C#):

1. A webapp 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 Azure Cosmos DB, and then uploading the image itself to blob storage. The webapp does not respond back to user until all of these steps are completed. In this assignment, we make uploading of an image asynchronous: The application responds back to the client immediately, while the upload is still in progress, and serverless functions complete the steps in the background to add the image to the metadata database after the upload is finished. We also add an approval process, where an image does not become available until it has been approved (Approval requires a new role of `Approver` in the access control).

We use 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.

The addition to the webapp from the previous assignment is an action in the images controller for approving images. This reads messages from the `approval-requests`

[1] You will need to install .NET 9.0 to run your code for this assignment locally. You will also need to update the `dotnet-ef` tool to this version of .NET:

```
$ dotnet tool update --global dotnet-ef
```

Also make sure you are running the latest version of the Azurite storage emulator. Because of a bug, you will have to manually specify the version of the Azurite Docker image:

```
$ docker run -p 10000:10000 -p 10001:10001 -p 10002:10002 \
      mcr.microsoft.com/azure-storage/azurite:3.33.0
```

queue and leaves output messages in the `accepted-images` and `rejected-images` queues. Details for these queues are listed in the `appsettings.json` and `appsettings.Development.json` files (The URI is for the storage account):

```
"Data": { ...
    "Queues": {
      "Uri": "...",
      "AccountName": "...",
      "VisibilityTimeout": 60,
      "MaxApprovalRequests": 32,
      "ApprovalRequestsQ": "approval-requests",
      "ApprovedImagesQ": "approved-images",
      "RejectedImagesQ": "rejected-images"
    }
```

The user secrets should be extended with the access key for queue storage, and the access key for the Azure storage account should be stored as a secret under the appropriate key in Azure Key Vault:

```
"Credentials": { ...
    "Queues": {
        "AccessKey": "..."
    }
```

For testing purposes, you will want to use the Consumption Plan for Azure Functions. As before, we use SQL Database to store user information, Azure Blob storage to store image files, and Azure Cosmos DB to store metadata about the images.

In the previous assignment, the webapp 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 outputs where appropriate. The annotations must specify containers and queue names, as well as connection strings that specify the details required to access those resources.

For example, the function that responds to uploads to blob storage is annotated as follows:

```
[Function(nameof(UploadResponder))]
[QueueOutput(...queue name...,
             Connection = ...connection string...)]
public string Run([BlobTrigger(...container name... + "/{blobname}",
                               Connection = ...connection string...)]
                   string myBlob,
                string blobname,
                IDictionary<string,string> metadata,
                FunctionContext _context)
```

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);
```

The webapp now contains an Approve function (required role: `Approver`) that allows someone to read the queue of approval request messages (for images whose uploading has completed) and decide whether to approve the image. Approved images are placed on a queue `approved-images`, and a serverless function processes the messages on that queue and add them to the image metadata database in CosmosDB:

```
[Function(nameof(ApproveResponder))]
public Image Run([QueueTrigger(...queue name...,
                              Connection = ...connection string...)]
                    QueueMessage message)
```

In theory, you can have this function store an image metadata object in Cosmos DB by annotating it with `CosmosDBOutput`, but we instead do this manually: A singleton Cosmos client is created in `Program.cs` and injected into the function when it is instantiated, and that is used to obtain a client for the Cosmos DB container in which metadata is stored (as in the previous assignment).

Another serverless function is triggered by the addition of messages to the `rejected-images` queue. It does not produce any output but deletes the image from blob storage, using a blob storage client that is created manually in the function.

For local development, the connection strings can be defined in the settings file `local.settings.json`. Playing the role of `appsettings.Development.json` for ASP.NET development, this should specify connection strings for the storage used when running in development mode:

```
{
  "IsEncrypted": false,
  "Values": {
    "BlobStorageConnectionString": "UseDevelopmentStorage=true",
    "QueueStorageConnectionString": "UseDevelopmentStorage=true",
    "CosmosDbConnectionString": "..."
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated"
  }
}
```

The connection string named `AzureWebJobsStorage` specifies access to the storage used by the Azure Functions runtime; `UseDevelopmentStorage=true` specifies use of the local Azurite emulator. The connection string for the Cosmos DB database is also stored in this settings file[2]. This is used in the main program to make a singleton Cosmos DB client available as a service to be injected where necessary:

```
var host = new HostBuilder()
    .ConfigureFunctionsWorkerDefaults()
    .ConfigureServices(services =>
    {
      CosmosClient imageDbClient = ApproveResponder.GetImageDbClient();
      services.AddSingleton<CosmosClient>(imageDbClient);
    })
    .Build();
```

This function generates the Cosmos DB Client by reading the connection string from an environment variable where it will have been loaded from the settings file:

```
public static CosmosClient GetImageDbClient()
{
    var connectionString =
          Environment.GetEnvironmentVariable(CosmosDbConnectionString);
    var client = new CosmosClient(connectionString);
    return client;
}
```

For development purposes, you should use the Azurite[3] local Azure storage emulator to test usage of the Blob and Queue storage locally. You can use SQL Server running in a container as in previous assignments while in development, though it is not needed by the serverless functions. It is up to you to use either the Cosmos DB emulator or Azure Cosmos DB for storing image metadata during development. You can use Azure Storage Explorer to create the queues in the Azurite storage emulator, and see the progress of an upload of an image to blob

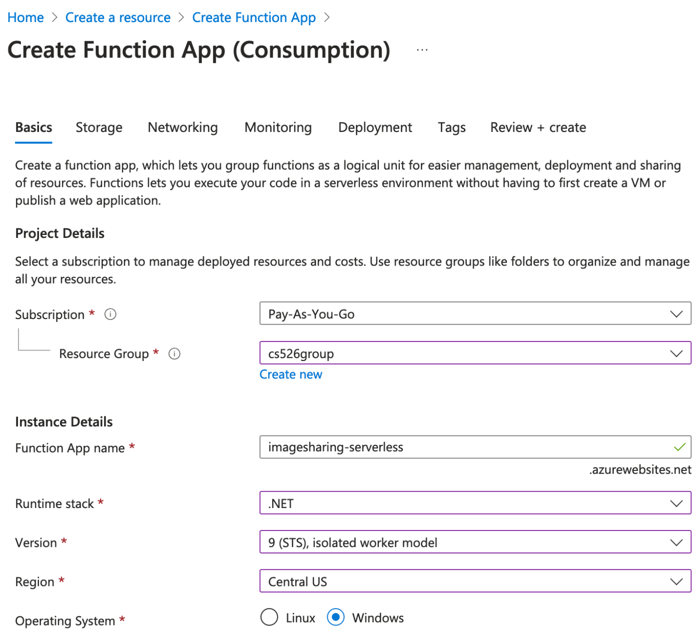
[2] It is obviously unfortunate that Azure Functions does not read from user secrets. The Azure Functions Core Tools CLI (below) includes an operation for encrypting this settings file: `func settings encrypt`. By default, the local settings are not packaged with the function when it is deployed.
[3] https://learn.microsoft.com/en-us/azure/storage/common/storage-use-azurite?tabs=visual-studio.

storage[4], then insertion of a request to approve the image, then (after approval is granted) insertion of a message into the approved-images queue, then finally insertion of the metadata record in Cosmos DB.

You can test your functions locally using the Azure Functions Core Tools CLI[5]. JetBrains Rider relies on this to run the functions in your project[6] or you may run the functions using the CLI yourself. Now you can run the webapp from JetBrains Rider, and the functions will be triggered by completion of upload of images and by approval of images in the webapp. The log from Azure Functions Core Tools CLI will record the triggers that launch executions of the functions in the background as the webapp is executing.

Deployment:

Once you have your application running, deploy your application in Azure App Service, and your functions in Azure Functions. Deploy the webapp as you did in the previous assignment, creating the Docker image and pushing it to Azure Container Service. Make sure that you have created the queues that you will need for the assignment, and that you have updated `appsettings.json` and Azure Key Vault with the information needed for accessing the queues. To deploy the functions, create an Azure Functions resource with consumption-based billing:



Create Function App (Consumption)

Create a function app, which lets you group functions as a logical unit for easier management, deployment and sharing of resources. Functions lets you execute your code in a serverless environment without having to first create a VM or publish a web application.

Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *: Pay-As-You-Go
Resource Group *: cs526group

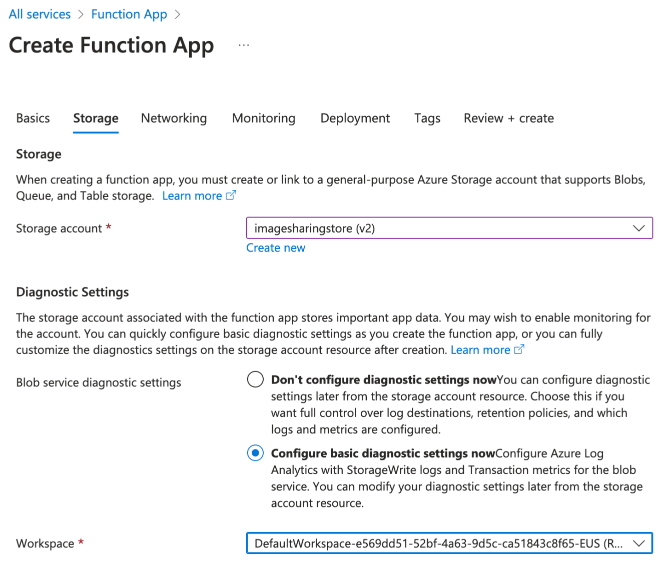
Instance Details

Function App name *: imagesharing-serverless .azurewebsites.net
Runtime stack *: .NET
Version *: 9 (STS), isolated worker model
Region *: Central US
Operating System *: Linux [✓ Windows]

---

[4] If you cannot see the uploaded blobs in Azure Storage Explorer, choose the view for "Active Blobs and Blobs Without Current Version."
[5] https://learn.microsoft.com/en-us/azure/azure-functions/functions-run-local?pivots=programming-language-csharp.
[6] https://blog.jetbrains.com/dotnet/2020/10/29/build-serverless-apps-with-azure-functions/.

Azure Functions needs access to storage for storing its own state; make this the same test storage account that you are using for development with blob and queue storage:


All services > Function App >

Create Function App ...

Basics | **Storage** | Networking | Monitoring | Deployment | Tags | Review + create

Storage

When creating a function app, you must create or link to a general-purpose Azure Storage account that supports Blobs, Queue, and Table storage. Learn more

Storage account *: imagesharingstore (v2)

Create new

Diagnostic Settings

The storage account associated with the function app stores important app data. You may wish to enable monitoring for the account. You can quickly configure basic diagnostic settings as you create the function app, or you can fully customize the diagnostics settings on the storage account resource after creation. Learn more

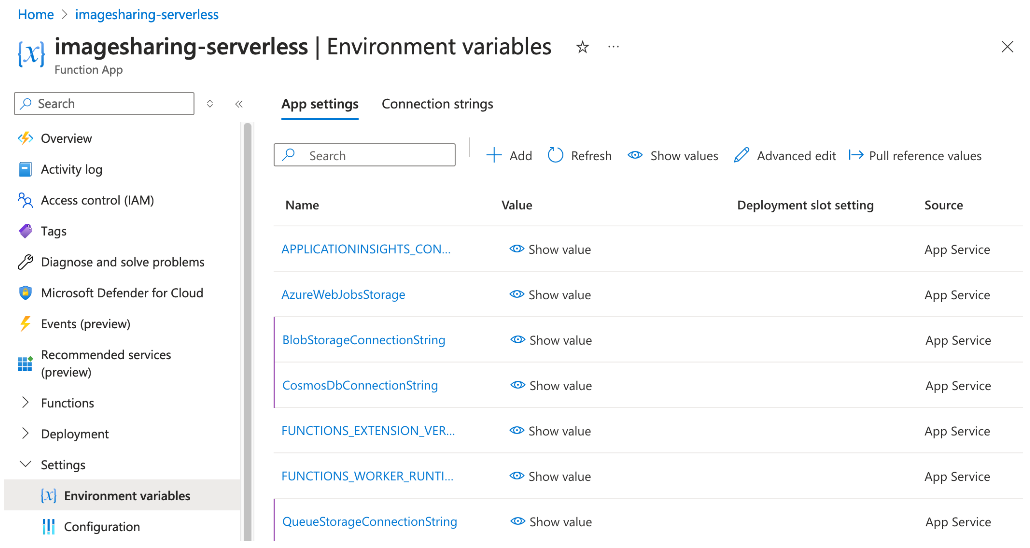
Blob service diagnostic settings:

- **Don't configure diagnostic settings now** You can configure diagnostic settings later from the storage account resource. Choose this if you want full control over log destinations, retention policies, and which logs and metrics are configured.
- [✓ **Configure basic diagnostic settings now** Configure Azure Log Analytics with StorageWrite logs and Transaction metrics for the blob service. You can modify your diagnostic settings later from the storage account resource.]

Workspace *: DefaultWorkspace-e569dd51-52bf-4a63-9d5c-ca51843c8f65-EUS (R...

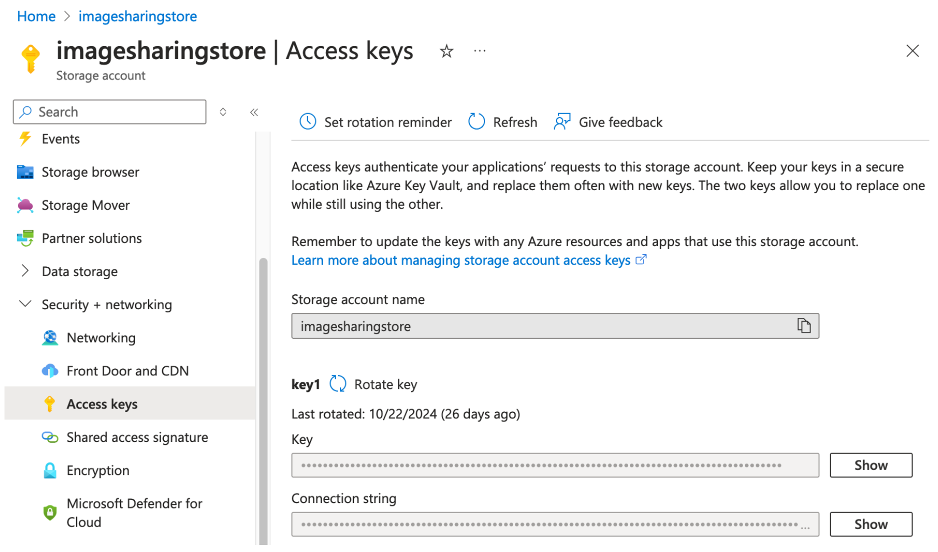
For other settings, you may want to enable application insights, and you need to allow public access (even though the functions are only triggered by changes to your Azure storage). You should disable continuous deployment since we are not implementing devops.

In the previous assignment, you stored passkeys in Azure Vault and relied on managed identity to allow access to these secrets, but Azure Functions is not well integrated with managed identity. So instead, the connection strings must be specified by environment variables in the context of the serverless functions:



Name	Value	Deployment slot setting	Source
APPLICATIONINSIGHTS_CON...	Show value		App Service
AzureWebJobsStorage	Show value		App Service
BlobStorageConnectionString	Show value		App Service
CosmosDbConnectionString	Show value		App Service
FUNCTIONS_EXTENSION_VER...	Show value		App Service
FUNCTIONS_WORKER_RUNTI...	Show value		App Service
QueueStorageConnectionString	Show value		App Service

You can get the connection string for blob and queue storage from the tab for the storage account access keys:



Access keys authenticate your applications' requests to this storage account. Keep your keys in a secure location like Azure Key Vault, and replace them often with new keys. The two keys allow you to replace one while still using the other.

Remember to update the keys with any Azure resources and apps that use this storage account.
Learn more about managing storage account access keys

Storage account name: imagesharingstore

key1 Rotate key
Last rotated: 10/22/2024 (26 days ago)
Key: •• Show
Connection string: •• Show

Once you have set application settings, deploy the functions as follows from the `ImageSharingFunctions` home directory[7]:

```
func azure functionapp publish yourserverlessappname
```

Now run the webapp as you did in the previous assignment. You should be able to view Azure storage using Azure 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 (and no record for that image in Cosmos DB), then see that image awaiting approval in the webapp, 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 webapp once approval is granted), and you can view this image in the list of all images in the webapp. Similarly, if an uploaded image is not approved, there should be no record inserted for it into Cosmos DB, but the uploaded blob deleted from blob storage once a message in the `rejected-messages` queue is processed by `RejectResponder`.

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 Humphrey_Bogart. This folder should contain a single folder for your solution

[7] https://learn.microsoft.com/en-us/azure/azure-functions/functions-run-local?pivots=programming-language-csharp&tabs=macos%2Cisolated-process%2Cnode-v4%2Cpython-v2%2Chttp-trigger%2Ccontainer-apps#project-file-deployment.

named `ImageSharingServerless`, with projects `ImageSharingWithServerless`. `ImageSharingFunctions` and `ImageSharingModels` (the latter providing some definitions used by both the webapp and the Azure functions app).

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 webapp. *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.