CS526 Enterprise and Cloud Computing
Stevens Institute of Technology—Fall 2024
Assignment Three—Cloud Storage

This assignment requires the use of JetBeans Rider. You are provided with an ASP.NET project (Version 8.0, C#). The structure of the presentation logic (default layout and styles) is the same as in the previous assignments. Your project includes the models from the previous assignment, and the basic UI. In this project, you will move the application you have developed in the last two assignments into the cloud. The external API as far as users of your system are concerned will not change over the previous assignments, (aside from on additional piece of functionality), but your application will now run as a cloud application. There are several Nuget packages that need to be added for the application:

`Microsoft.Azure.Cosmos`: Client for Azure Cosmos DB.
`Azure.Storage.Blobs`: Client for Azure Blob Storage.
`Azure.Data.Tables`: Client for Azure Table Storage.
`Microsoft.AspnetCore.Identity.EntityFrameworkCore`: ASP.NET Identity.
`Microsoft.EntityFrameworkCore.SqlServer`: EF for SQL Server.
`Microsoft.EntityFrameworkCore.Tools`: Tools for EF.
`Microsoft.EntityFrameworkCore.Cosmos`: LINQ extensions for Cosmos DB.
`Microsoft.AspnetCore.Diagnostics.EntityFrameworkCore`: Help with database errors.
`Microsoft.Extensions.Logging.AzureAppServices`: Logging for Azure App Service.
`Azure.Extensions.AspNetCore.Configuration.Secrets`: Using Key Vault for secrets.
`Azure.Identity`: Authenticating to Key Vault.

Users: SQL Database

For user management, we will store information about users and roles in **SQL Database**, a cloud-based sharded version of SQL Server[1]. Your main task here will be defining a connection string to use SQL Database. You will continue to store the user (and role) tables in this database, inheriting from `IdentityContext`. You should be careful to set up SQL Database in the *serverless tier*, otherwise you may incur substantial charges for the use of Azure. Beware that using SQL Database is not the same experience as with SQL Server. You may need to deal with transient failures, particularly due to cold start of a database in the serverless tier. The SQL client tries to mask these failures, if you configure it to retry the database connection if it fails, but you may still have to deal with these issues[2]:

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>
{
    // For SQL Database, allow for db connection sometimes being lost
    options.UseSqlServer(connectionString, options =>
                                           options.EnableRetryOnFailure());
    ...
});
```

[1] Note that it is also possible to use the .NET Identity API with Cosmos DB instead, using the `AspNetCore.Identity.CosmosDb` library which uses the Entity Framework interface to Cosmos DB.
[2] For example, the default connection timeout in SQL Database is 30 seconds. You may want to modify the connection string to increase this to say 180 seconds, so the connection does not timeout while clearing the database during initialization.

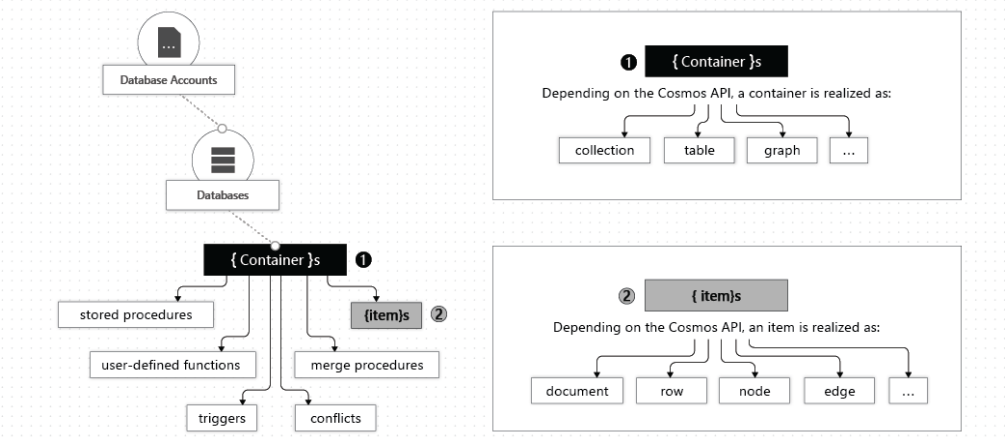
Since the SQL database schema has changed, you will have to rebuild the database for this assignment. Make sure to drop the database from the previous assignment, delete any migrations, then create a new migration for this assignment and recreate the database. See below for more on how to do this.

Images: Azure Cosmos DB and Blob Storage

We no longer store image metadata in SQL Server, but instead in **Azure Cosmos DB**, using the Cosmos DB client API. We encapsulate access to Cosmos DB using a repository with this interface:

```
public interface IImageStorage
{
    public Task InitImageStorage();

    public Task<string> SaveImageInfoAsync(Image image);

    public Task<Image> GetImageInfoAsync(string UserId, string Id);

    public Task<IList<Image>> GetAllImagesInfoAsync();

    public Task<IList<Image>> GetImageInfoByUserAsync(ApplicationUser user);

    public Task UpdateImageInfoAsync(Image image);

    public Task RemoveImageAsync(Image image);

    public Task RemoveImagesAsync(ApplicationUser user);

    ...
}
```

The following illustrates the various concepts that make up the API for Cosmos DB:



The implementation of the repository for image storage obtains a client for the container you have defined for storing image metadata. Best practice is for a single instance of a Cosmos DB client to be used in an application; therefore, we add this client as a singleton service to be injected where needed in the application:

```
CosmosClient imageDbClient =
    new CosmosClient(imageDbUri, imageDbAccessKey, cosmosClientOptions);
builder.Services.AddSingleton<CosmosClient>(imageDbClient);
```

Here is an example of the client being injected into the repository for image storage:

```
public ImageStorage(..., CosmosClient imageDbClient, ...) {
    ...
    Database imageDbDatabase = imageDbClient.GetDatabase(imageDatabase);
    Container imageDbContainer = imageDbDatabase.GetContainer(imageContainer);
```

When storing the metadata for an image, we must generate the value for the primary key (a field that **must** be called `id` when serialized to the database). For the partition key, we use the primary key of the user that uploaded the image, so the items for all images for a user are stored in the same partition. We also store the name of the user who uploaded the image, as an example of the Subset Pattern:

```
using Newtonsoft.Json;
public class Image
{
    [JsonProperty(PropertyName = "id")]
    public string Id { get; set; }
    ...
    public string UserId { get; set; }
    public string UserName { get; set; }
    ...
}
```

When we insert a record into Cosmos DB, we must separately specify the partition key:

```
image.Id = Guid.NewGuid();
Image item = await databaseContainer.CreateItemAsync<Image>(
                                item: image,
                                partitionKey: new PartitionKey(image.UserId));
```

When retrieving an individual image's record, we require both the partition key and the record's primary key. The LINQ extension for Cosmos DB also provides a `WithPartitionKey` extension method that allows all the images for a particular partition to be queried. This means that the only image query that does not go to just one partition is that for all images.

We use **Azure Blob Storage** to save images, instead of storing them on the server file system. We extend the API for the image storage repository to encapsulate the details of going to blob storage:

```
public interface IImageStorage
{
    ...

    public Task SaveFileAsync(IFormFile imageFile,
                              string userId, string imageId);
```

```
    public Task RemoveImageFileAsync(string userId, string imageId);

    public string ImageUri(string userId, string imageId);
}
```

The implementation of this API obtains a client for the blob container you have defined for storing images as blobs:

```
BlobServiceClient blobServiceClient =
    new BlobServiceClient(imageStorageUri, credential, null);
BlobContainerClient blobContainerClient =
    blobServiceClient.GetBlobContainerClient(storageContainer);
```

Logs: Azure Table Storage

In this application, we will log every time someone views an image. Add a table in **Azure Table Storage** that records every time that an image and its details are viewed, and by whom. Call this table `imageviews`. Assume that the most common query is "What images were viewed today?" so you can use the same key organization as in the example in the lectures. Use a partition key based on date, and use a primary key based on image id and date. There is an additional action in the `Images` controller, called `ImageViews`, that displays a report of viewing (optionally just for today), organized by image id (display each image caption in the report as well). Create a separate role, called Supervisor, to authorize access to this action.

We encapsulate access to the logs with a repository, with this as its API:

```
public interface ILogContext
{
    public Task AddLogEntryAsync(string user, ImageView imageView);

    public AsyncPageable<LogEntry> Logs(bool todayOnly);
}
```

The implementation of this API obtains a client to store and retrieve logs in table storage, using `LogEntry` as the entity model for log entries:

```
TableClient tableClient = new TableClient(logTableServiceUri,
                                          logTableName,
                                          credential);
```

Development

For development purposes, you can continue to use MS SQL Server in a container for the user database, and you should use the Azurite[3] local Azure storage emulator to test usage of the Blob and Table storage locally. You can pull a Docker image:

```
docker pull mcr.microsoft.com/azure-storage/azurite:latest
docker run -p 10000:10000 -p 10001:10001 -p 10002:10002 \
    --name azurite mcr.microsoft.com/azure-storage/azurite
```

[3] https://learn.microsoft.com/en-us/azure/storage/common/storage-use-azurite?tabs=docker-hub .

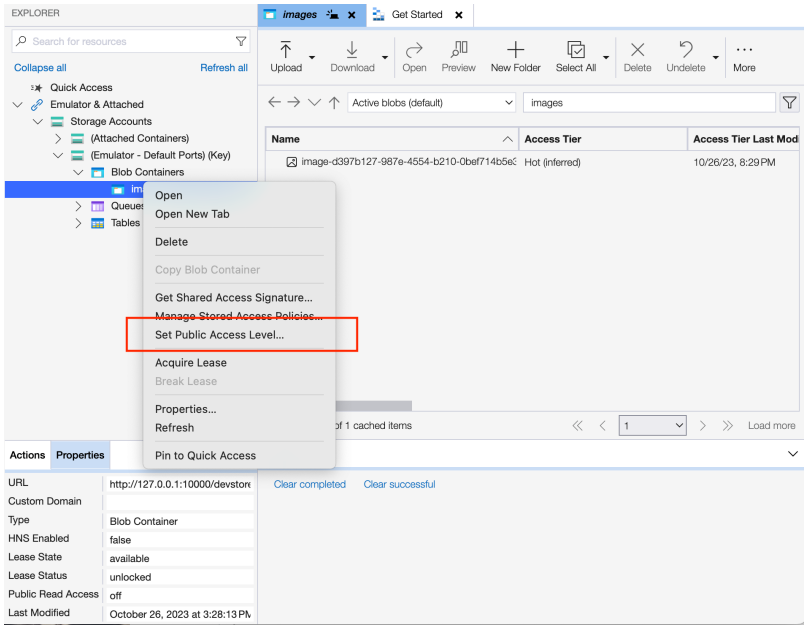
The `docker run` command binds to ports 10000 (Blob storage), 10001 (Queue storage) and 10002 (Table storage) on the local machine. The development account and password for Azurite services are:

Account Name: `devstoreaccount1`
Account Key:
`Eby8vdM02xNOcqFlqUwJPLlmEtlCDXJ1OUzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPTOtr/KBHBeksoGMGw==`

Use Azure Storage Explorer[4] to create the requisite containers and tables in Azurite:

1. For blob storage, you should create a container called `images`.
2. For table storage, you should create a table called `imageviews`.
3. For Azure Cosmos DB, you should create a database called `imagesharing` and a container called `images`, either in the emulator or in Azure Cosmos DB depending on which you are using.

You will need to allow public access to images uploaded with Azure. To do this, use Azure Storage Explorer to configure public access:



For Cosmos DB, one possibility is to use the Cosmos DB storage emulator[5]. However, it requires some setup (You need to install its self-signed SSL certificate as a trust anchor on your machine) and it is not supported for ARM machines. You may find it just as easy to use it (in a serverless deployment) in Azure, see the next section.

[4] https://azure.microsoft.com/en-us/products/storage/storage-explorer.
[5] https://learn.microsoft.com/en-us/azure/cosmos-db/how-to-develop-emulator?pivots=api-nosql&tabs=windows%2Ccsharp.

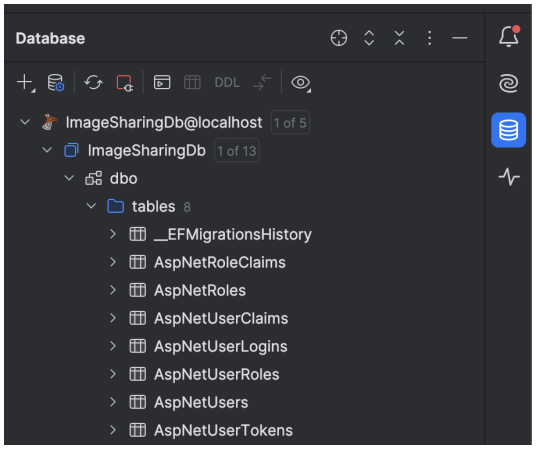
The connection information for the various forms of storage is in the application settings descriptor, for example, for the emulators in development mode[6]:

```
"Data": {
  "ApplicationDb": {
    "ConnectionString": "Server=localhost;...",
    "Database": "ImageSharingDb"
  },
  "ImageDb": {
    "Uri": "https://localhost:8081", // For emulator
    "Database": "imagesharing",
    "Container": "images"
  },
  "ImageStorage": {
    "Uri": "http://127.0.0.1:10000/devstoreaccount1",
    "AccountName": "devstoreaccount1",
    "Container": "images"
  },
  "LogEntryDb": {
    "Uri": "http://127.0.0.1:10002/devstoreaccount1",
    "AccountName": "devstoreaccount1",
    "Table": "imageviews"
  }
}
```

When using `dotnet-ef` to set up the database tables, set the environment variable that ensures it executes in development mode, so it accesses the local database and retrieves the credentials from `secrets.json`:

```
dotnet ef migrations add Initial -- --environment Development
dotnet ef database update -- --environment Development
```

Use the Database tool in JetBrains Rider (`View | Tool Windows | Database`) to create a connection to the database and verify that the ASP.NET Identity tables have been created:



[6] Because of issues with the Cosmos DB emulator, you may find it easier to develop using Azure Cosmos DB in the cloud, creating an Azure Cosmos DB database there as described in the section on deployment. For all other storage (SQL Server, Blob Storage and Table Storage), use local emulators as these application settings assume.

Note that the Cosmos DB emulator requires access with HTTPS, while the emulator for Blob and Table storage only supports HTTP (HTTPS is *essential* for production use with Azure storage). The connection string for `ApplicationDb` will connect to MS SQL Server running in a docker container on your local machine, as in the previous assignment. To organize access to connection strings, the class below provides keys for paths in the application settings JSON document:

```
public class StorageConfig
{
    /*
     * Keys for user database metadata.
     */
    public const string ApplicationDbConnString =
                              "Data:ApplicationDb:ConnectionString";
    public const string ApplicationDbUser = "Credentials:ApplicationDb:User";
    public const string ApplicationDbPassword = "Credentials:ApplicationDb:Password";
    public const string ApplicationDbDatabase = "Data:ApplicationDb:Database";
    /*
     * Keys for image database (Cosmos DB) metadata.
     */
    public const string ImageDbUri = "Data:ImageDb:Uri";
    public const string ImageDbAccountName = "Data:ImageDb:AccountName";
    public const string ImageDbAccessKey = "Credentials:ImageDb:AccessKey";
    public const string ImageDbDatabase = "Data:ImageDb:Database";
    public const string ImageDbContainer = "Data:ImageDb:Container";
    /*
     * Keys for image storage (Blob) metadata.
     */
    public const string ImageStorageUri = "Data:ImageStorage:Uri";
    public const string ImageStorageAccountName = "Data:ImageStorage:AccountName";
    public const string ImageStorageAccessKey = "Credentials:ImageStorage:AccessKey";
    public const string ImageStorageContainer = "Data:ImageStorage:Container";
    /*
     * Keys for log storage (Table) metadata.
     */
    public const string LogEntryDbUri = "Data:LogEntryDb:Uri";
    public const string LogEntryDbAccountName = "Data:LogEntryDb:AccountName";
    public const string LogEntryDbAccessKey = "Credentials:LogEntryDb:AccessKey";
    public const string LogEntryDbTable = "Data:LogEntryDb:Table";
    /*
     * URI for key vault with credentials for databases.
     */
    public const string KeyVaultUri = "Data:KeyVault:Uri";
}
```

The settings are loaded and available as configuration information at database build time (using `builder.Configuration`), for example:

```
string dbConnectionString =
          builder.Configuration[StorageConfig.ApplicationDbConnString];
```

The configuration information is also available while the application is running, injecting an instance of type `IConfiguration` into a controller or repository. For example, the image storage repository injects configuration information to obtain the connection information and credentials for blob storage:

```
public ImageStorage(IConfiguration configuration, ...) {
```

```
...
Uri imageStorageUri =
   new Uri(configuration[StorageConfig.ImageStorageUri]);

string imageStorageAccountName =
   configuration[StorageConfig.ImageStorageAccountName];
string imageStorageAccountKey =
   configuration[StorageConfig.ImageStorageAccessKey];
StorageSharedKeyCredential credential =
   new StorageSharedKeyCredential(imageStorageAccountName,
                                  imageStorageAccountKey);

BlobServiceClient blobServiceClient =
   new BlobServiceClient(imageStorageUri, credential, null);
```

Note that the access keys for authenticating to storage are **not** stored in the application descriptor. We will see below how to store this information securely for development and for production. The keys for storage specify a separate configuration section in a JSON object with a key "`Credentials`" that stores these access keys.

The connection information for development includes the account name and account key for authenticating to a storage server emulator. The account keys for the emulators are well known and can be found from the documentation for Azurite and the Cosmos DB emulator. It would clearly not be acceptable to store critical credentials such as account keys in the application source code. Instead, we should use the Secret Manager in ASP.NET to store confidential information outside the application, and use Azure Key Vault to look up this information when the application is deployed in the cloud.

Secret Manager[7] is used during development to store secrets in a special file outside the project, with the file path specified by a user secrets id provided in the project file:

```
<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
  <UserSecretsId>...</UserSecretsId>
</PropertyGroup>
```

Here is a sample version of the `secrets.json` file:

```
{
  "Credentials": {
    "ApplicationDb": {
      "User": "...",
      "Password": "..."
    },
    "ImageDb": {
      "AccessKey": "..."
    },
    "ImageStorage": {
      "AccessKey": "..."
    },
    "LogEntryDb": {
      "AccessKey": "..."
```

[7] https://learn.microsoft.com/en-us/aspnet/core/security/app-secrets?view=aspnetcore-8.0&tabs=windows.

```
    }
  }
}
```

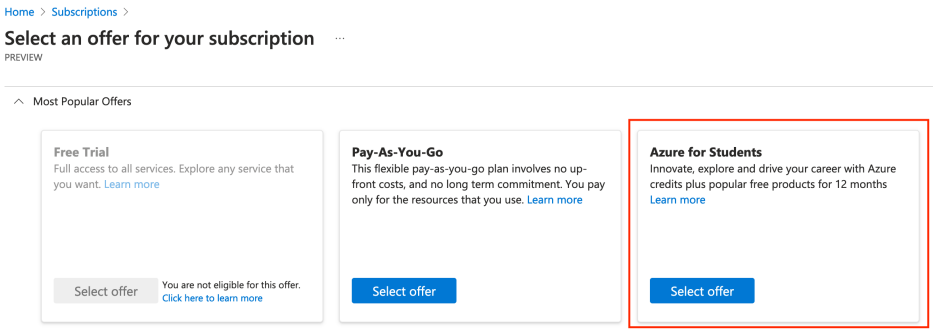
You should never edit this file directly (It may be encrypted in a future version of JetBrains Rider). You can edit the secrets file as a JSON document in JetBrains Rider, as in the previous assignment.

During development, the configuration API reads connection information from the development application settings and reads the secret credential from secrets storage:

```
public const string ImageDbUri = "Data:ImageDb:Uri";
public const string ImageDbAccessKey = "Credentials:ImageDb:AccessKey";
...
string imageDbUri = configuration[StorageConfig.ImageDbUri];
string imageDbAccessKey = configuration[StorageConfig.ImageDbAccessKey];
CosmosClient imageDbClient =
   new CosmosClient(imageDbUri, imageDbAccessKey, cosmosClientOptions);
```

Deployment

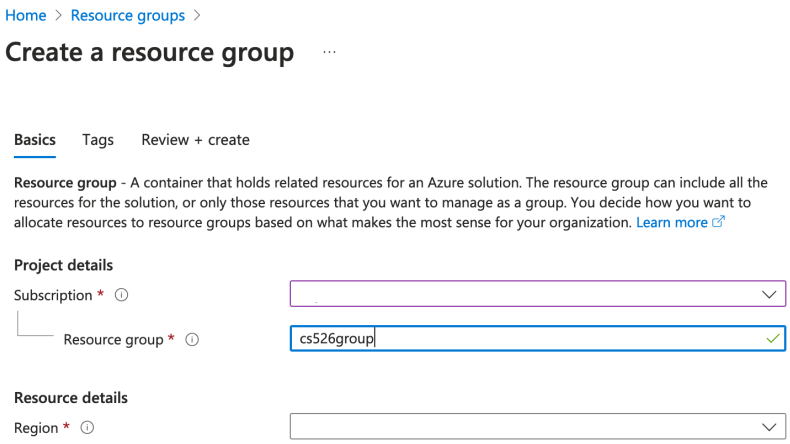
To deploy the app in Azure, you will need to create an account at the Azure portal[8], and then create the resources your application will be using (See the lectures for more information). First add a subscription to Azure for Students, which you are eligible for if you sign in with an academic email address. This will grant you some free credits for usage of Azure:



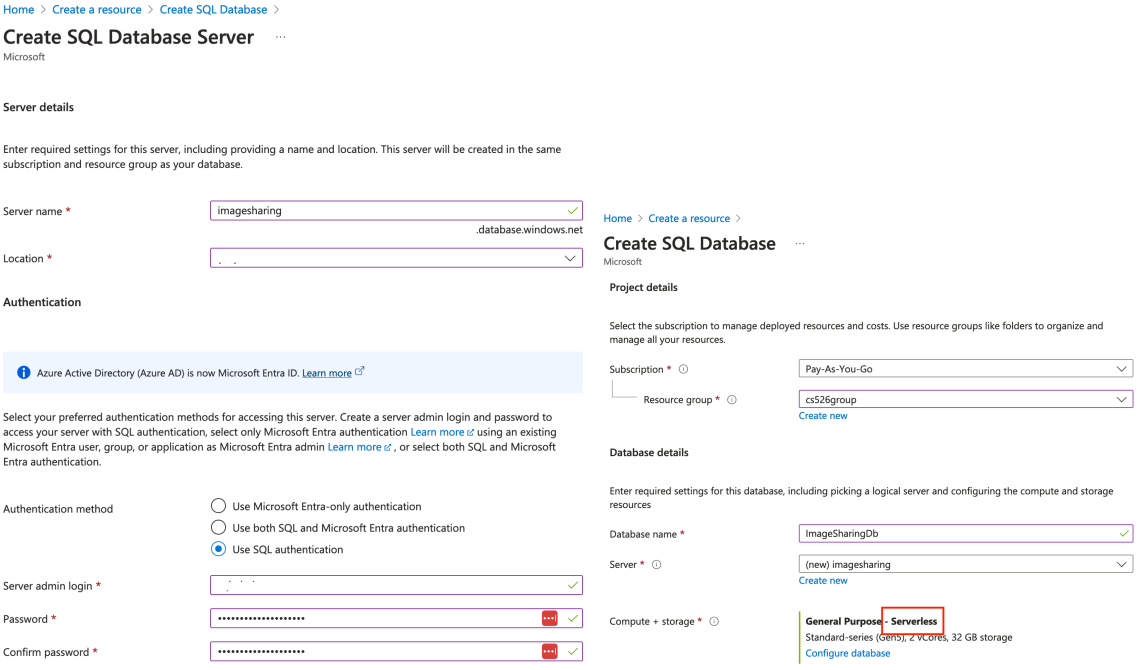
Create a resource group, e.g., `cs526group`. All of the resources that you create will be in this resource group. Make sure to create all your resources in the same region[9].

[8] https://azure.microsoft.com/en-us/features/azure-portal/.

[9] Do **not** use US East, since some resources are not available there at the free tier, such as App Services deployed from containers.

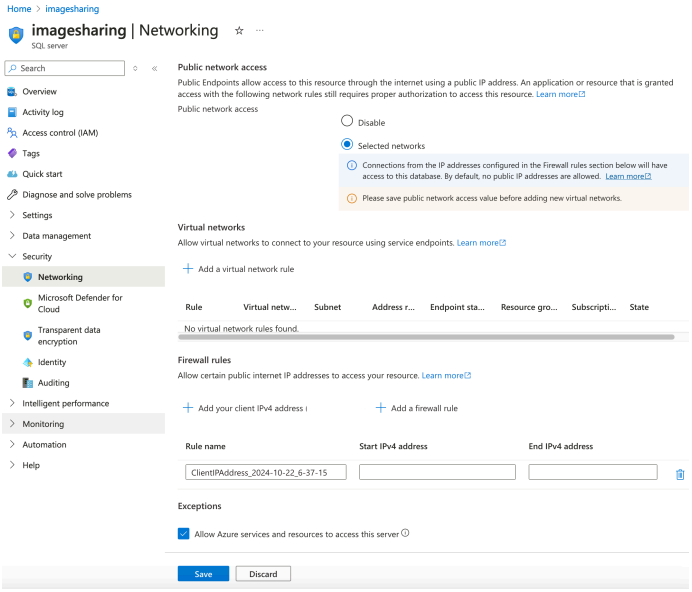


Create a SQL Database in that resource group, e.g., `ImageSharingDb`. You will need to set up a logical server that serves as a central administrative point for a collection of databases (Note that it is **not** a SQL Server instance), and the name of the server must be unique because it is used for the host name for the server, *`yourservername`*`.database.windows.net`. Use SQL authentication for the server and create the credentials for a database user:



Make sure that you create the server in the serverless tier. This should mean that you will only be charged a nominal amount for running the database, but you should monitor the database's running costs so you are not surprised. You may have to delete the database after demonstrating its working if you notice surprising cost behavior.

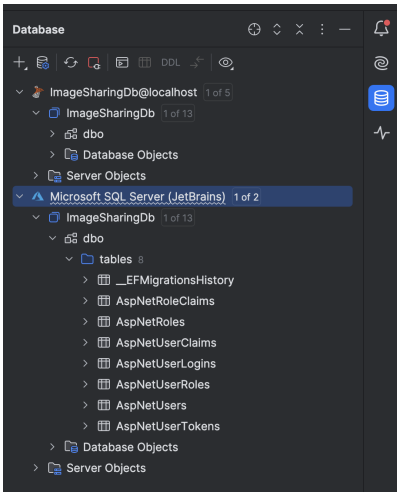
You will want to allow access to the database from your application running in Azure App Services (You will also want to allow access from your IP address, for example to set up the database using `dotnet ef`):



Once you have created the SQL database in Azure, you need to set up the database schema using `dotnet-ef`. If you set up `appsettings.Development.json`. to reference the database in SQL Database, then you can put the credentials for accessing the cloud database in `secrets.json`. You can then run the `dotnet-ef` commands, to set up the initial migration and initialize the database, in development mode, so it will retrieve the credentials from `secrets.json`, as we showed in the previous section:

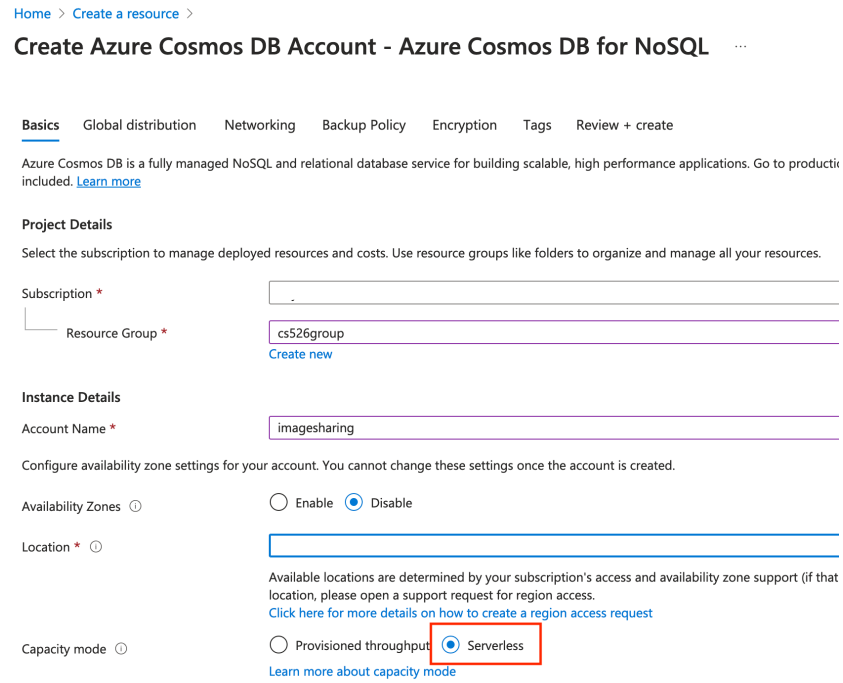
```
dotnet ef migrations add Initial -- --environment Development
dotnet ef database update -- --environment Development
```

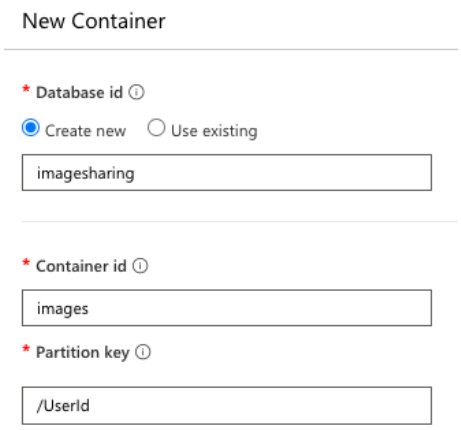
Note that if you are giving the database a “cold start” (since it is serverless), the connection attempt may timeout when you first try to perform the migration. You can again use the JetBrains Rider Database tool to verify that the tables have been created:



When you deploy your app in production mode, it will obtain the credentials from Azure Key Vault (See below). You should **never** put the credentials in `appsettings.json` or `appsettings.Development.json`.

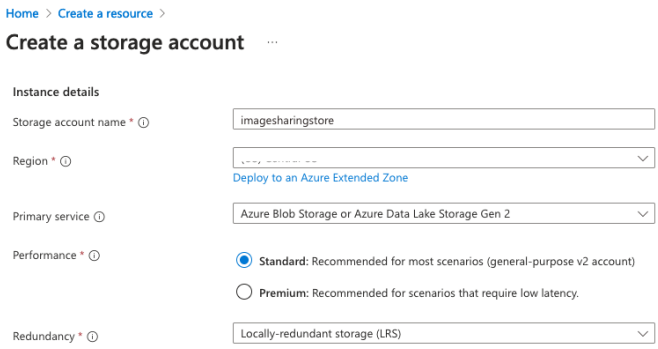
Create an Azure Cosmos DB database, using the API for NoSQL. Make sure it is in the same location as the other resources, e.g., East US, and **make sure you create it in the serverless tier**. You will need to specify a unique account name; it is used to identify the URL for the database server using virtual host *youraccountname*`.documents.azure.com`. Once you have created the account, you can then create the database (namespace) and the container within that database that holds a document (metadata record) for each image that is uploaded:





Note that you will also have to specify the partition key for documents in that container.

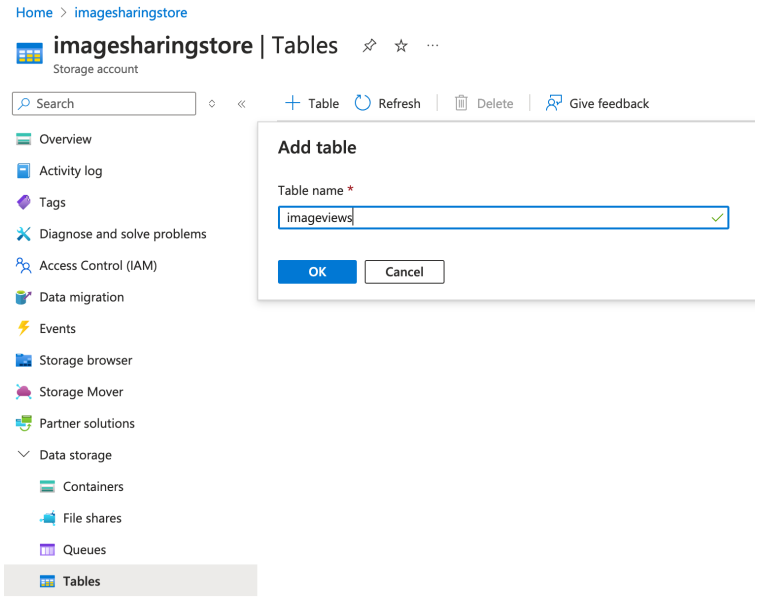
To load images themselves, create a storage account, with a different (unique) account name, and with locally redundant storage only to minimize cost:



Once you have created the storage account, you can create a container in blob storage (e.g., called `images`) for uploading image files. It will be available at this URL[10]:

`https://`*`youraccountname`*`.blob.core.windows.net/images`

For logs, we have two options. We could also use Azure Cosmos DB for storing logs, this time using the API for Table storage when you create the account. For production purposes, this is the better option, giving more options for controlling throughput and for global distribution. However, we will instead create a table under the storage account that we have created; this is the more economical option because we are only charged for the storage we actually use, rather than the storage provisioned.

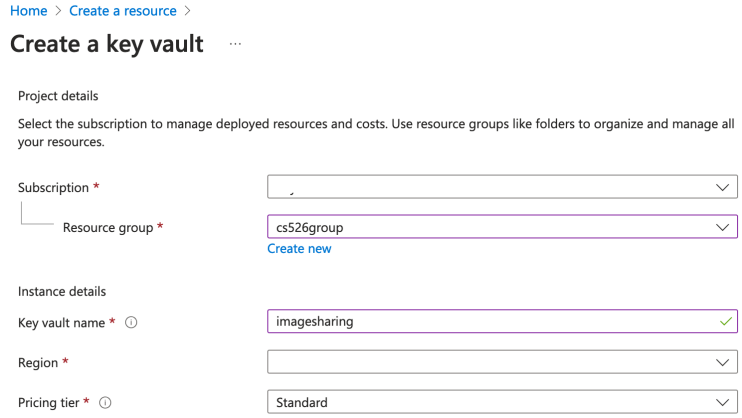


The .NET client transparently handles either this Table storage or the API provided by Cosmos DB. Assuming you created a table with name `imageviews`, it is exposed at the URL:

[10] When instantiating `BlobServiceClient` in `ImageStorage`, do **not** include the context root with the container name in the URI for the blob service.

`https://youraccountname.table.core.windows.net/imageviews`

Once you have created these resources, you can obtain the connection information, including endpoint URLs, account names and account keys. You should fill in the default application settings descriptor with the connection information (but **not** access keys). For production deployment in Azure, `secrets.json` is not available for storing passwords and access keys, and you should **never** store sensitive information such as credentials in the source code. Instead, you will store these credentials in Azure Key Vault[11], which you should again create in the same region as the other resources.



Once you have created the key vault, it will be available at the URI:

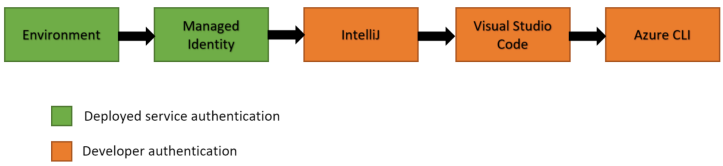
`https://yourvaultname.vault.azure.net`

Store this URI in the application settings descriptor. At build time, the settings are loaded and made available as configuration information to be injected where needed into the storage repositories[12]:

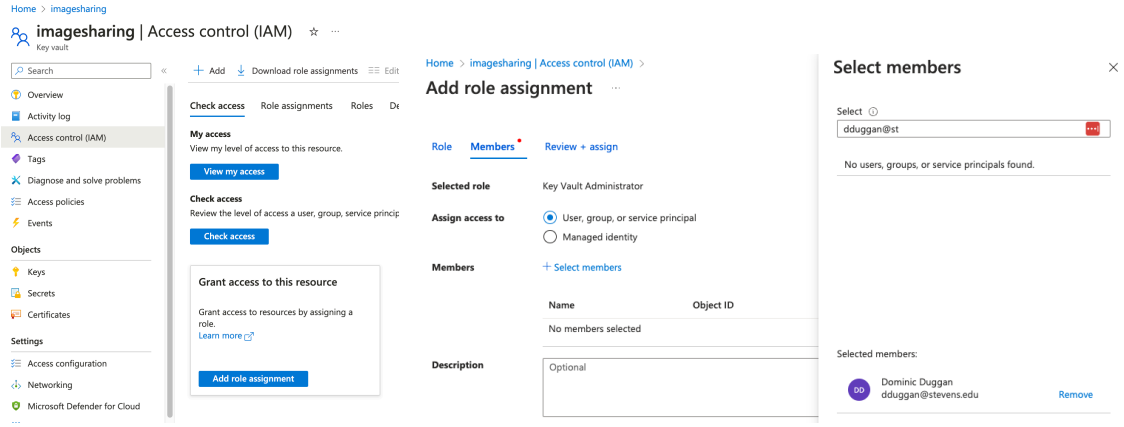
```
if (builder.Environment.IsProduction())
{
    string vault = builder.Configuration[StorageConfig.KeyVaultUri];
    Uri vaultUri = new Uri(vault);
    builder.Configuration.AddAzureKeyVault(vaultUri,
                                           new DefaultAzureCredential());
}
```

[11] https://learn.microsoft.com/en-us/aspnet/core/security/key-vault-configuration?view=aspnetcore-8.0.

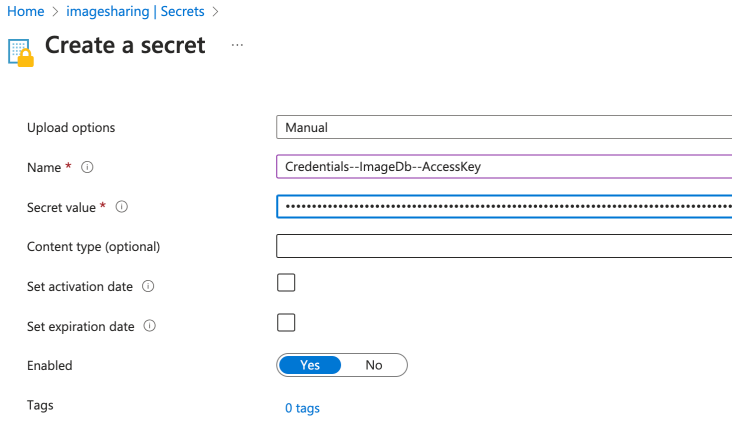
[12] `DefaultAzureCredential` attempts to authenticate through various possible sources of credentials, starting with environment variables, then checking if the app is running on an Azure host with Managed Identity enabled. You will assign a role to the identity for the app that allows it to read the key vault.



To store secrets in the key vault, you must first give yourself permission to manage secrets there[13]. In Azure Portal, select the Access Control (IAM) tab for the key vault you have created, then assign a role to yourself of Key Vault Administrator:



This will allow you to create new secrets in the key vault, e.g.:



Note that although the separator in a path in a configuration key is colon (":", e.g., "`Credentials:ImageDb:AccessKey`"), the separator for the key in Azure Key Vault is double-dash ("--", e.g., "`Credentials--ImageDb--AccessKey`"). This double-dash is automatically converted to colon as the secrets are loaded from Key Vault into the application configuration. This is the list of secrets you will need to store in Azure Key Vault:

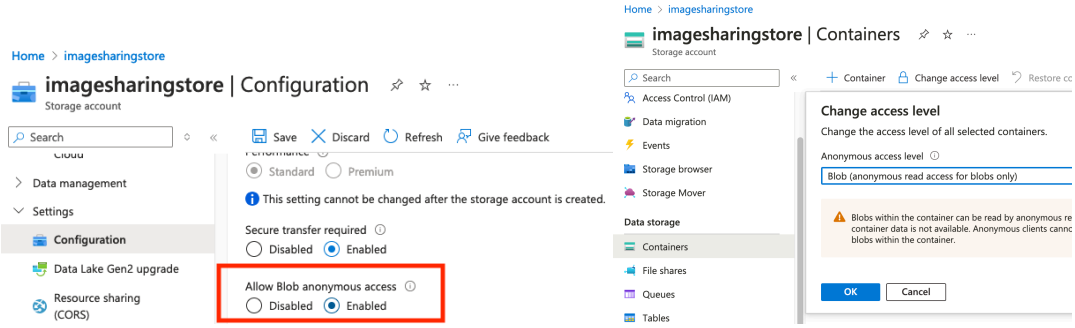
- `Credentials--ApplicationDb--User`
- `Credentials--ApplicationDb--Password`
- `Credentials--ImageDb--AccessKey`
- `Credentials--ImageStorage--AccessKey`
- `Credentials--LogEntryDb--AccessKey`

As with the Azurite emulator, to have images display in a browser after uploading to Azure Blob Storage, you will need to configure your blob storage to allow anonymous blob access[14].

---

[13] https://learn.microsoft.com/en-us/azure/role-based-access-control/role-assignments-portal?tabs=delegate-condition.

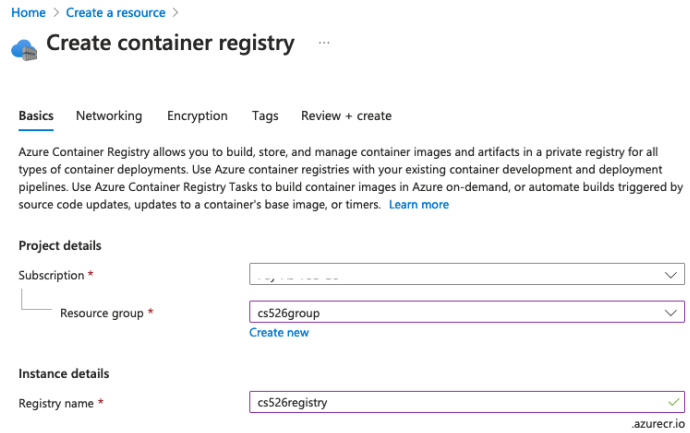
[14] https://learn.microsoft.com/en-us/azure/storage/blobs/anonymous-read-access-configure?tabs=portal .



We would prefer to place restrictions on access to blob storage, e.g., defining a private virtual network for the storage and only allowing access from your own IP address, but this would prevent the webapp from accessing the storage. This problem can be solved by creating a private endpoint on the network for the storage and associating the app with this endpoint, but we do not pursue this any further[15].

To deploy the Web app, you will find it useful to install the Azure CLI[16], since it provides some functionality that is not available at the Azure Portal interface, e.g., purging deleted Azure Key Vault secrets that may be interfering with the re-addition of those secrets.

We will deploy the app as a container, using App Service. First create a container registry using the Azure portal (Here we are using the registry name `cs526registry`, and we are using the resource group `cs526group` already created):



Next, in the project folder, use the `dotnet` command line tool, installed when you installed .NET, to publish the contents of the folder:

```
dotnet publish -c Release --no-self-contained
               -o ~/tmp/cs526/ImageSharingWithCloud/publish
cd ~/tmp/cs526/ImageSharingWithCloud
```

[15] https://stackoverflow.com/a/73970554.
[16] https://learn.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest.

Use an editor of your choice to create a file, in the directory where you have published the compiled app, called `Dockerfile` with these lines[17]:

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS runtime
WORKDIR /app
COPY publish .
ENV ASPNETCORE_HTTP_PORTS=80
EXPOSE 80
ENTRYPOINT ["dotnet", "ImageSharingWithCloud.dll"]
```

Now build an image of the application you are going to deploy:

```
docker build -t cs526/imagesharingcloud .
```

Instead of running the container locally, use the Azure CLI you installed earlier to login to the private container registry you have created, then tag the image you created in your local registry and push it to your Azure registry:

```
az login
az acr login --name your-registry-name
docker tag cs526/imagesharingcloud
                    your-registry-name.azurecr.io/imagesharingcloud
docker push your-registry-name.azurecr.io/imagesharingcloud
```

In the overview page for your container registry in the Azure portal, you should see (under Services) the repository for the Docker image `imagesharingcloud` that you have pushed. The container registry in Azure has an admin account, disabled by default, which you will need to enable to deploy the containerized app in App Services[18]:

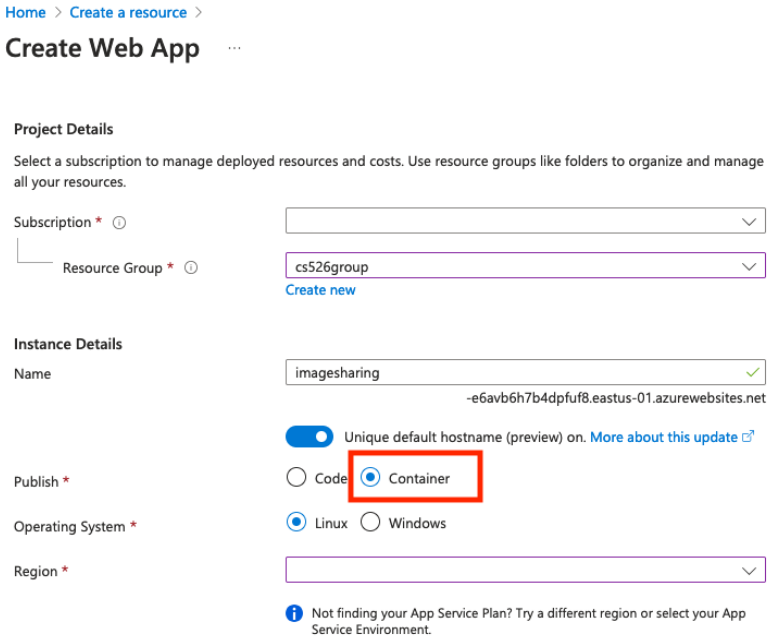
```
az acr update -n your-registry-name --admin-enabled true
```

Next, go to the Azure portal and create a webapp with App Service. Make sure to specify the same region for deployment as your other resources, the subscription for students (Azure For Students) and the same resource group as you are using for the other resources. **Be sure to pick a pricing plan other than the Premium Plan that Azure chooses as default.**

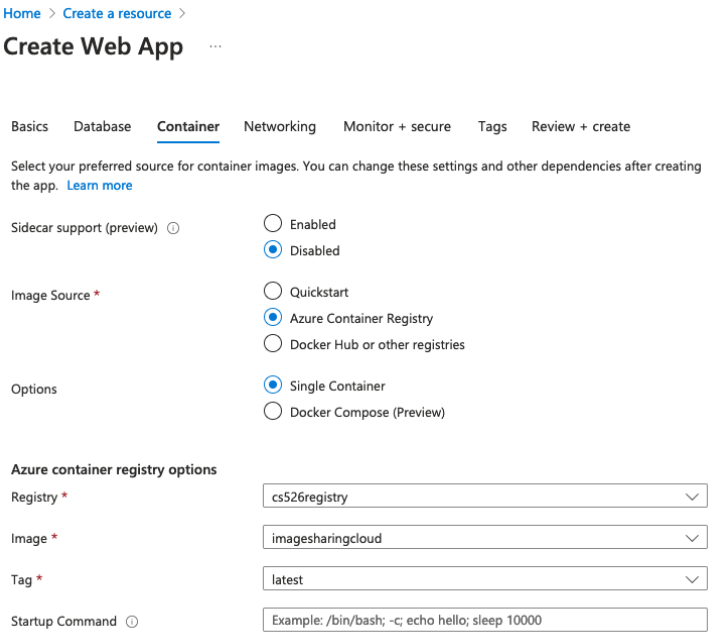
---

[17] App Service assumes that containers provide their HTTP service at port 80, but .NET by default provides its service at port 8080, to allow apps to run with non-root privileges. Therefore, we set ASPNETCORE_HTTP_PORTS in the container environment to override the .NET default.

[18] https://learn.microsoft.com/en-us/azure/container-registry/container-registry-authentication?tabs=azure-cli#admin-account.



Specify the image you have pushed to Azure Container Registry in the Container tab:

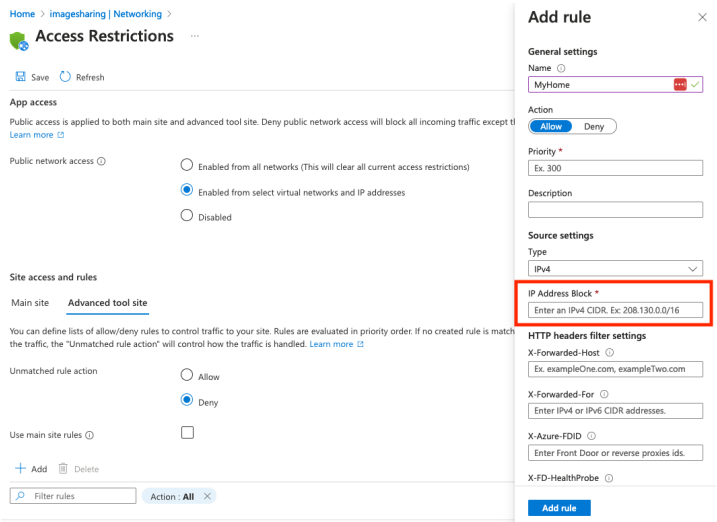


The URL for the app is based on the app name, the region where it is deployed, and a random hash:

`https://yourappname-randomhash.region.azurewebsites.net`

The URL window in your browser will show you that you have deployed in Azure App Service, and once you upload an image, the URLs of the images will show they are being hosted in Azure Blob Storage.

You should restrict access to this Web app to your own IP address. Once you have created the App Service resource, in the Settings | Networking tab, click on the configuration of Public Network Access and under Access Restrictions, select "Enabled from select IP addresses", set the default rule to Deny, and add another rule that only grants access at your IP address[19] (You can find it with a Google search, do not use the IP address of your network interface behind a NAT). Remember to press the "Save" button in the upper left!



You will need to create a managed identity for the app, and then give it permission to access the key vault. Use the Azure CLI to obtain a list of your subscriptions (You can also find your subscriptions at the Azure portal). Note the id for your "Azure For Students" subscription, and set this to be your default subscription:

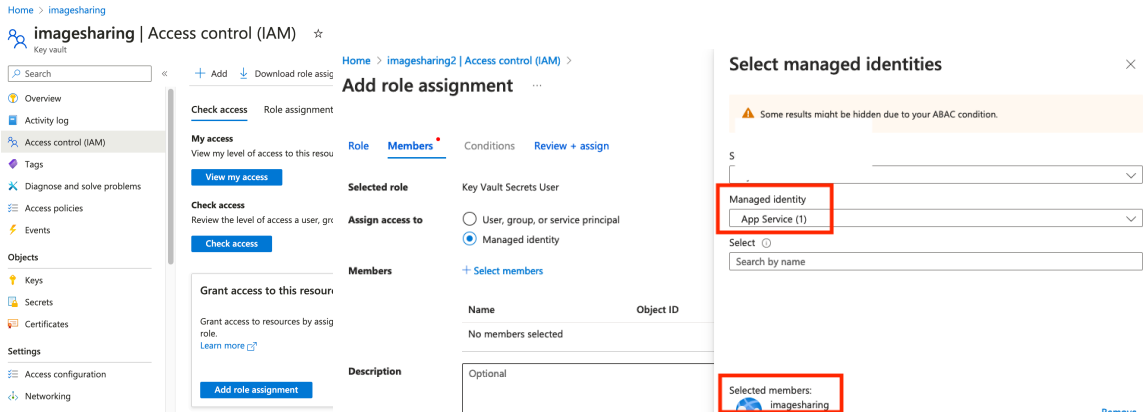
```
az login
az account set -s subscription-id
```

[19] If you are on-campus, use the address block for all Stevens external IP addresses.

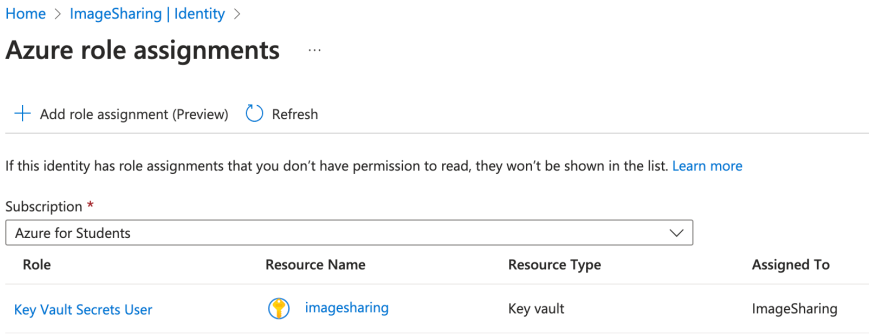
Now use the Azure CLI to create a managed identity for your app:

```
az webapp identity assign --name yourappname
          --resource-group resource-group-name
```

This returns a JSON document that includes a field with the principal id under which the app runs. In the Azure portal, go to the Azure Key Vault page and click on the Access Control (IAM) tab. You will add the role of Key Vault Secrets User for the managed identity of the Web app, just as you granted yourself the role of Key Vault Administrator, by clicking on Add Role Assignment:

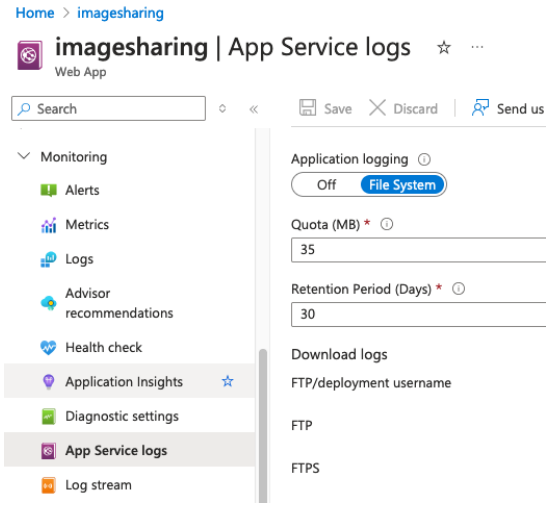


You can check the role assignments under the Identity tab for the App Service:

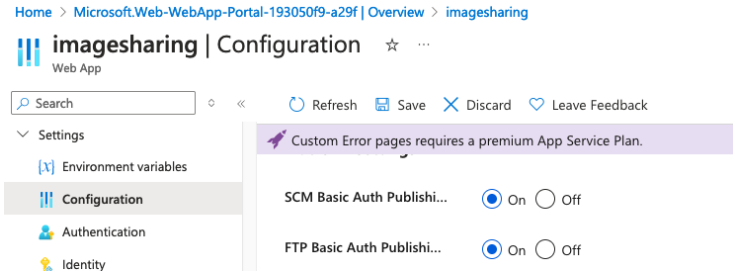


You should enable saving of logs[20] to the file system. Go to Monitoring | App Service Logs and check for application logging to the file system:

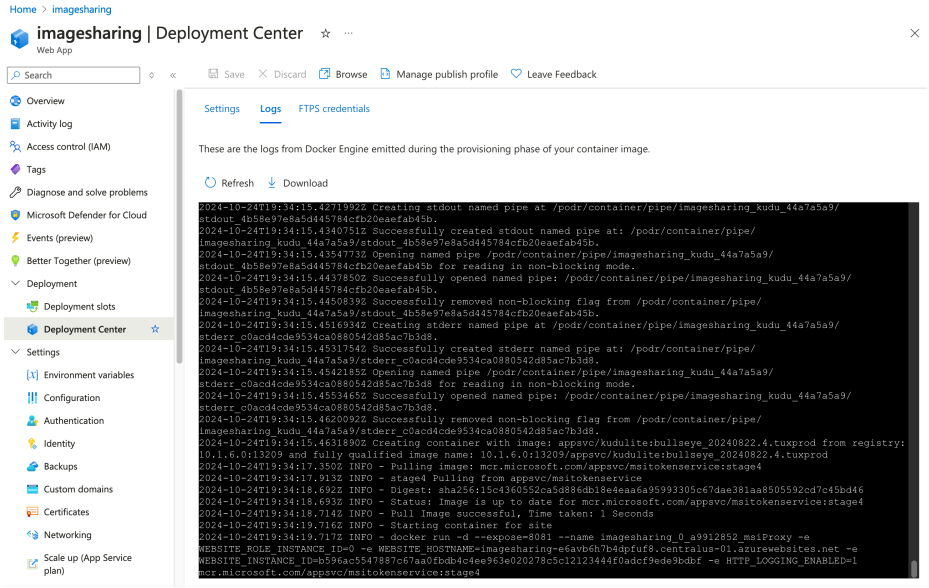
[20] https://learn.microsoft.com/en-us/azure/app-service/troubleshoot-diagnostic-logs.

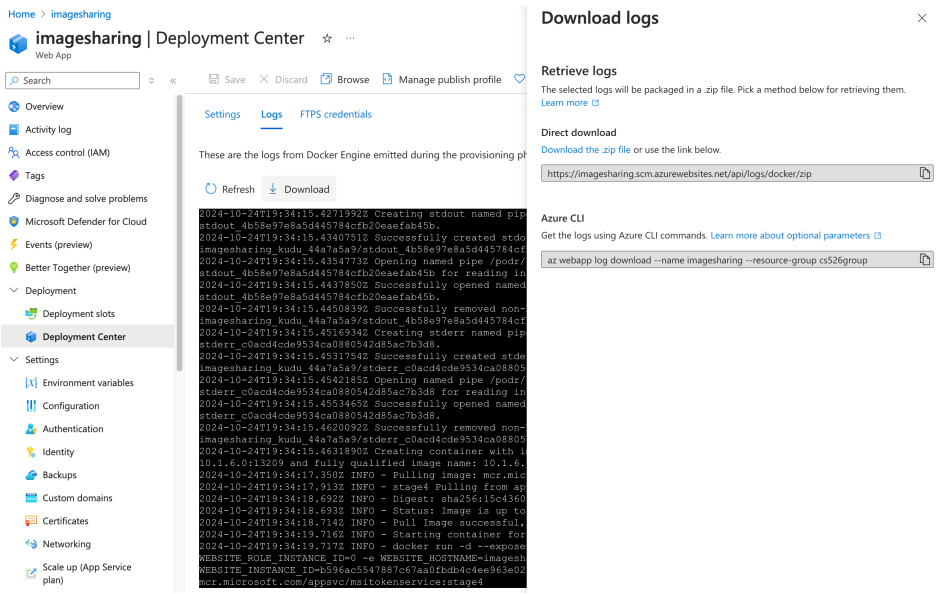


To view logs, you should enable basic authentication for the webapp, so you can browse the file system where the logs are stored:

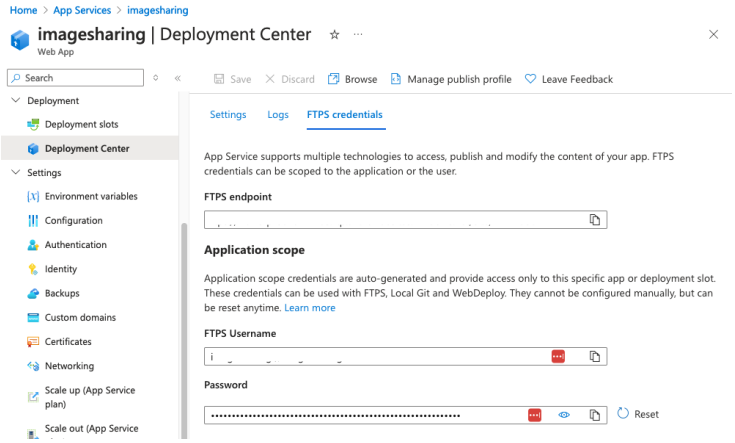


Go the deployment center to see the logs from when the container was provisioned,

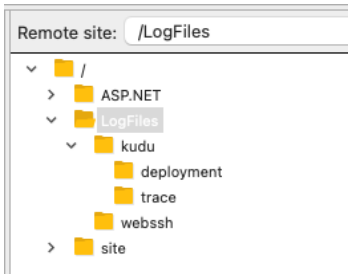




The deployment center also provides the ftp endpoint where container logs can be downloaded, as well as the credentials for authenticating at the endpoint[21]:



You can then use an ftp client such as FileZilla to view the logs, using the above credentials:



[21] https://learn.microsoft.com/en-us/azure/app-service/deploy-configure-credentials?tabs=portal#appscope.

You cannot run a container for the webapp on your local machine, using Azure storage, because it does not have access to the secrets in Azure Key Vault. It must run in Azure, with a managed identity that has been given permission to read the Key Vault.

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 named `ImageSharingWithCloud`.

In addition, record mpeg (MP4) videos demonstrating your deployment working. See the rubric for further details of what is expected. **Make sure that your Stevens username (e.g., `dduggan`) appears in the names of the resources (databases, containers, Web app, key vault) that you create in Azure.** 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. In addition to showing the app running in Azure, provide the application log of your demonstration. *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.