Lab 1 - Build an Azure Function

Functions have been the basic building blocks of software since the first lines of code were written, and the need for code organization and reuse became a necessity. Azure Functions expand on these concepts by allowing developers to create "serverless", event-driven functions that run in the cloud and can be shared across a wide variety of services and systems, uniformly managed, and easily scaled based on demand. Also, you can write Azure Functions in a variety of languages, including C#, JavaScript, Java, Python, Bash, and PowerShell, and they're perfect for building apps and nano-services that employ a compute-on-demand model.

In this lab, you will create an Azure Function that monitors a blob container in Azure Storage for new images, and then performs an automated analysis of the images using the Microsoft Cognitive Services Computer Vision API. Individually, The Azure Function will analyze each picture that is uploaded to the container for adult or racy content and create a copy of the image in another container. Images that contain adult or racy content will be copied to one container, and images that do not include adult or racy content will be reproduced to another. Also, the scores returned by the Computer Vision API will be stored in blob metadata.

Objective

In this hands-on lab, you will learn how to:

- Create an Azure Function App
- Write an Azure Function that uses a blob trigger
- Add application settings to an Azure Function App
- Use Microsoft Cognitive Services to analyze images and store the results in blob metadata

Prerequisites

The following are required to complete this hands-on lab:

- An active Microsoft Azure subscription.
- Microsoft Azure Storage Explorer (optional)

Steps

To build the solution in this lab, you have to follow the steps described in this section. From a high-level view the steps are:

- Create a resource group with a name like **rg-azurethursday-<initials>-<lab_no> (optional)**
- Provision a Storage Account
- Provision a Function App
- Add the Azure Function

Lab duration: **45 minutes**.

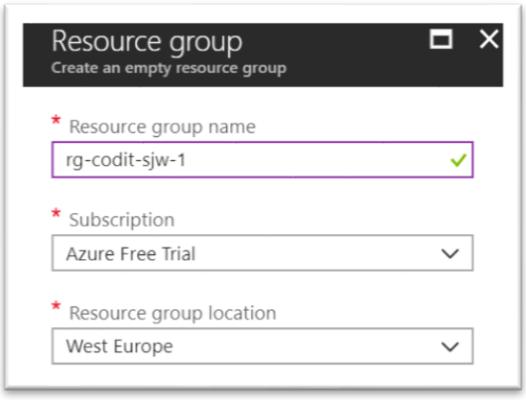
Created by Steef-Jan Wiggers, Microsoft Azure MVP, Codit Domain Lead Azure

Contact steef-jan.wiggers@codit.eu or twitter @steefJan

Step 1 - Create a resource group

The very first step in this lab is creating a resource group in your Azure subscription. A resource group is a logical container that groups all your resources. After the lab is finished, and you do not want to keep the resources, you can delete the resource group, and the Azure Resource Manager will remove all the resources for you.

1. In the Azure Portal navigate to Resource Groups in the left menu pane.
2. Click the **+ Add**.
3. Provide a name for the resource group (**rg- azurethursday -<initials>-<lab_no>)**, specify a Subscription, and a location.



4. Finally, click **Create** and a resource group will be created for you.
5. In the top right corner, a pop-up will appear, which you can click to go to your resource group.

Step 2 – Provision a Storage Account

Within the resource group, you can quickly add various types of Azure Resources. For this lab, we will need a storage account (blob) to upload an image to a container.

1. Go to the resource group you created earlier (Step 1).
2. Click **+ Add**.
3. A new pane will appear, where you can search for a resource (service).
4. Enter: *Storage Account*.
5. *Storage account - blob, file, table, queue* will appear.
6. Click the icon named Storage account – blob, file, table, queue.
7. A new pane will appear, where you can click **Create**.
8. Again a new pane will appear, and here you can start specifying a few properties for your Storage Account.
9. In the screenshot below, you will see the details you need to specify.

Create storage account

Basics | Advanced | Tags | Review + create

Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below. Learn more

PROJECT DETAILS

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

* Subscription: Azure Free Trial

* Resource group: rg-codit-labs-sjw

Create new

INSTANCE DETAILS

The default deployment model is Resource Manager, which supports the latest Azure features. You may choose to deploy using the classic deployment model instead. Choose classic deployment model

* Storage account name: storagesjw

* Location: West Europe

Performance: Standard / Premium

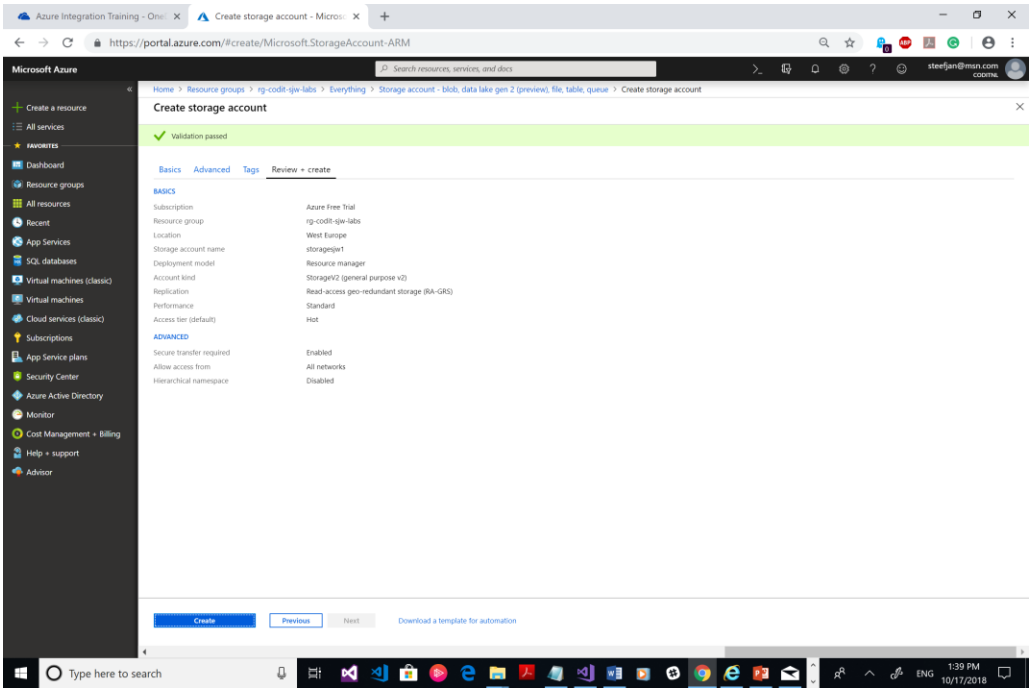
Account kind: StorageV2 (general purpose v2)

Replication: Read-access geo-redundant storage (RA-GRS)

Access tier (default): Cool / Hot

Review + create | Previous | Next : Advanced >

10. Choose a useful unique name (**storage<intials><labno>)**.
11. Click **Create + Review** once finishing specifying.



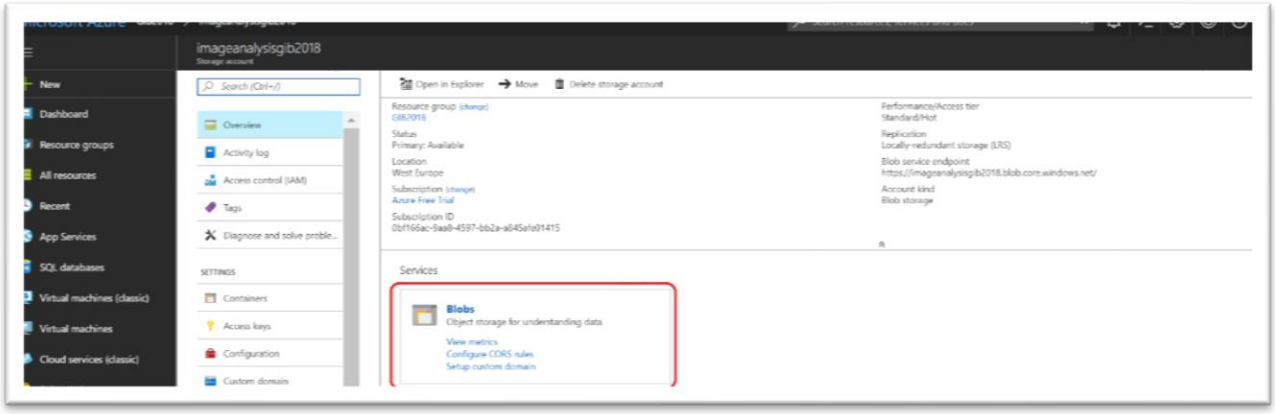
12. Click **Create**.

    Note: Keep every Azure resource in the same location to prevent unnecessary network charges for traffic between regions.

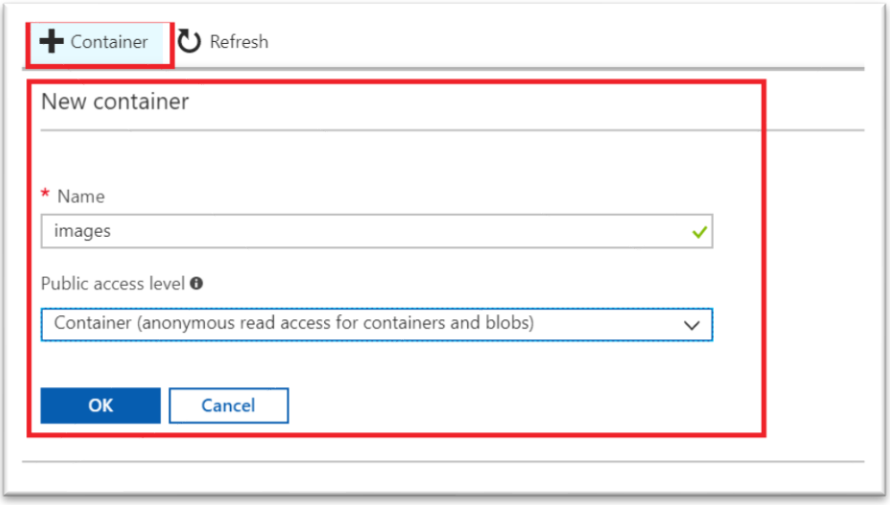
Step 3 – Create a container

Once the Storage Account (blob) is available for you, the next step is to add a container.

1. Go to the resource group you created earlier (Step 1).
2. Click the Storage Account you created in step 2.
3. Click **Blobs**.



4. Click on **+ Container**.
5. Specify images as the name for the container in the window that will appear.

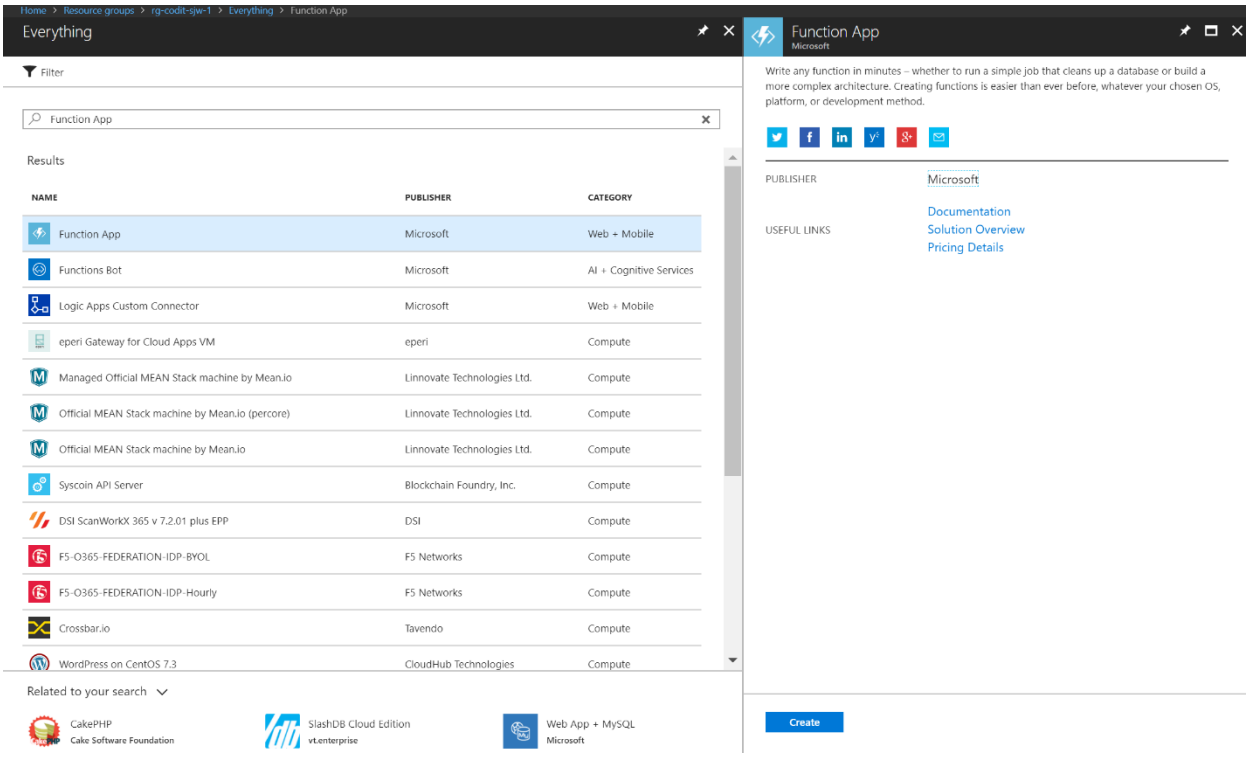


6. Change the public access level to read access for containers and blobs only. This access level is necessary to get access to your blob. If you keep it private, then you cannot find the blob and give you an **HTTP 404 Not Found**.
7. Click **OK**.

Step 4 – Provision a Function App

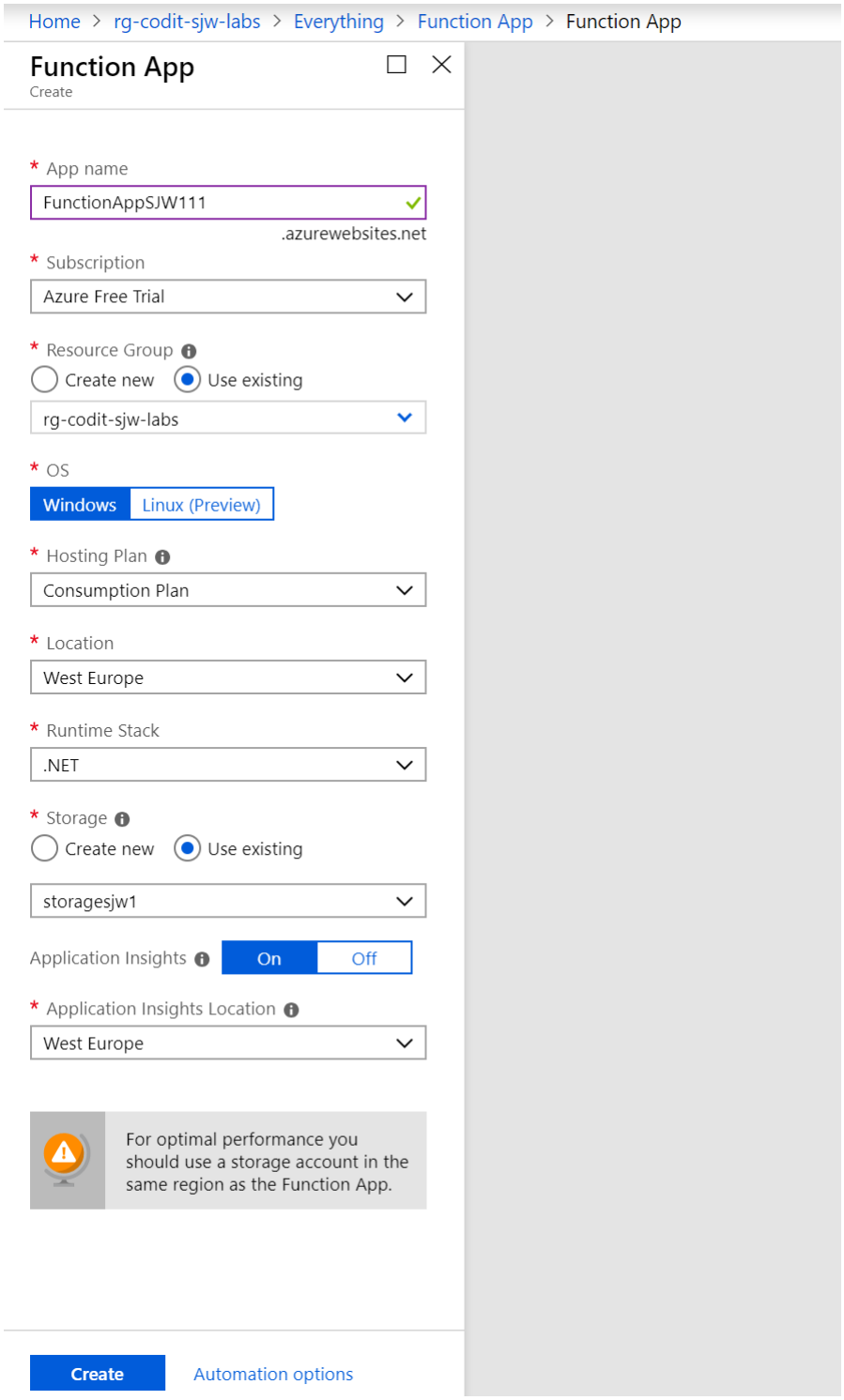
The first step in writing an Azure Function is to create an Azure Function App. In this exercise, you will create an Azure Function App using the Azure Portal. Subsequently, you will add the three blob containers to the storage account that is created in step 2: one to store uploaded images, a second to store images that do not contain adult or racy content, and a third to include images that *do* contain adult or racy content.

1. Open the Azure Portal in your browser. If you are asked to log in, please do so using your Microsoft account.
2. Go to your resource group. Click **+ Create a resource**, followed by **Function App** in the search window. Select Function App.

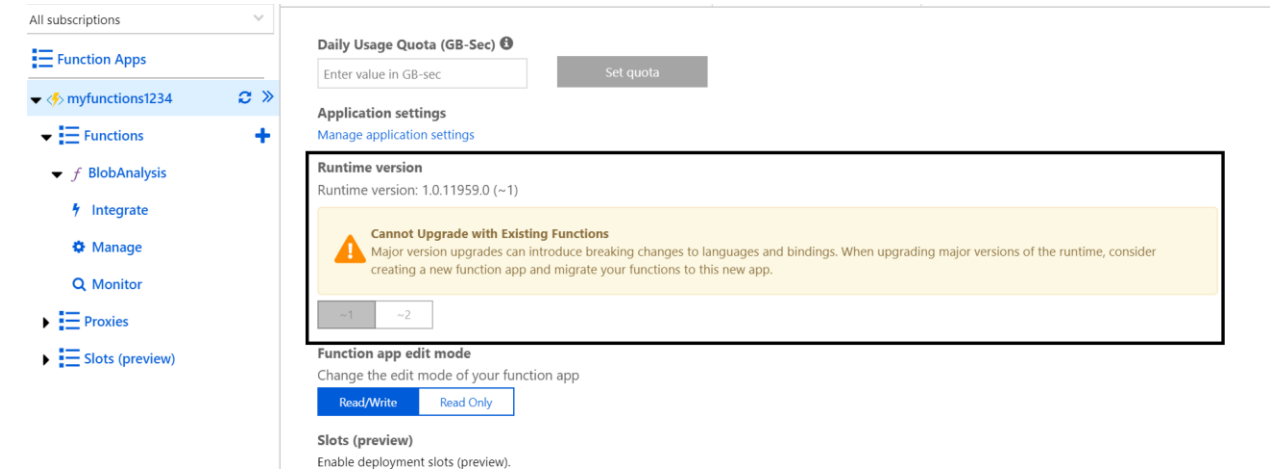


3. Enter an app name that is unique within Azure. Under **Resource Group**, select **Existing** to create a resource group for the Function App. Choose the **Location** nearest you, and accept the default values for all other parameters. Also, choose the existing storage account created in previous step. Then click **Create** to create a new **Function App**.

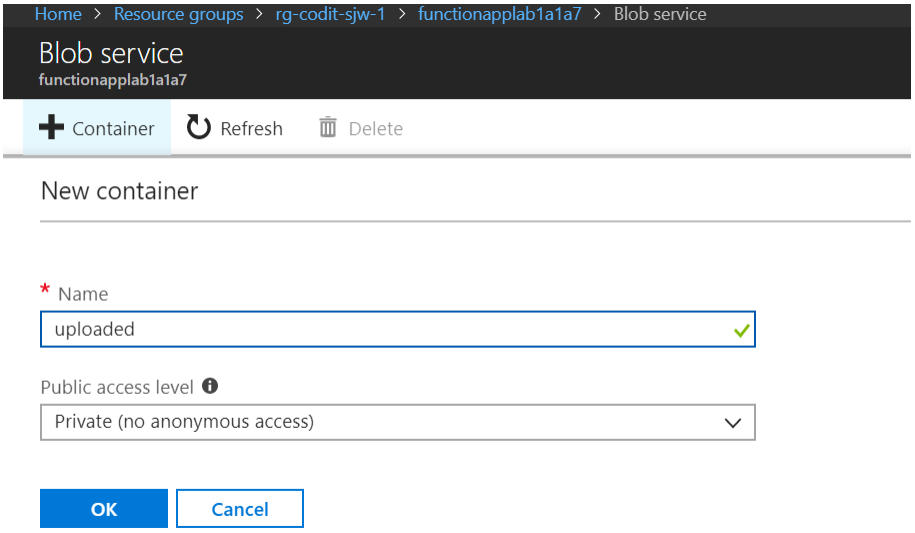
   *The app name becomes part of a DNS name and therefore must be unique within Azure. Make sure a green check mark appears to the name indicating it is unique. You probably **won't** be able to use "functions lab" as the app name.*



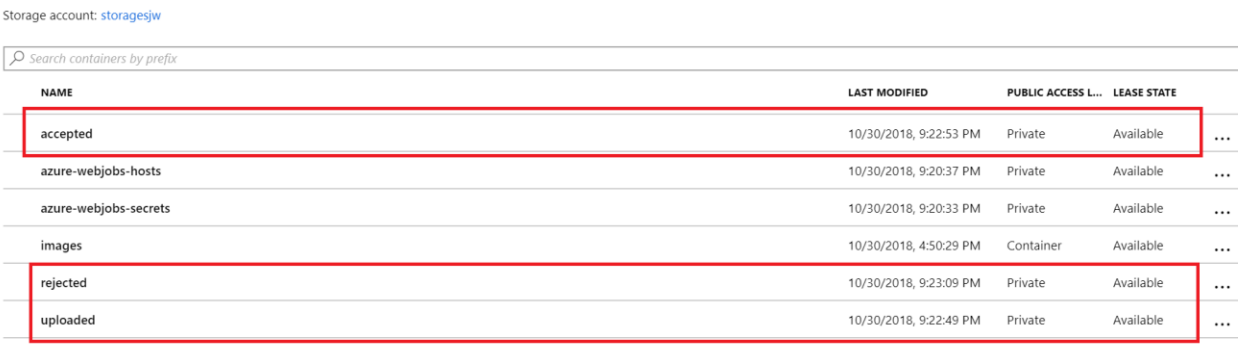
4. Open Application Settings and set the runtime to **V1**.



5. Select the storage account.
6. Click **Blobs** to view the contents of blob storage.
7. Click **+ Container**. Type "*uploaded*" into the **Name** box and set the **Public access level** to **Private**. Then click the **OK** button to create a new container.



8. Repeat Step 7 to add containers named "*accepted*" and "*rejected*" to blob storage.
9. Confirm that all three containers were added to blob storage.



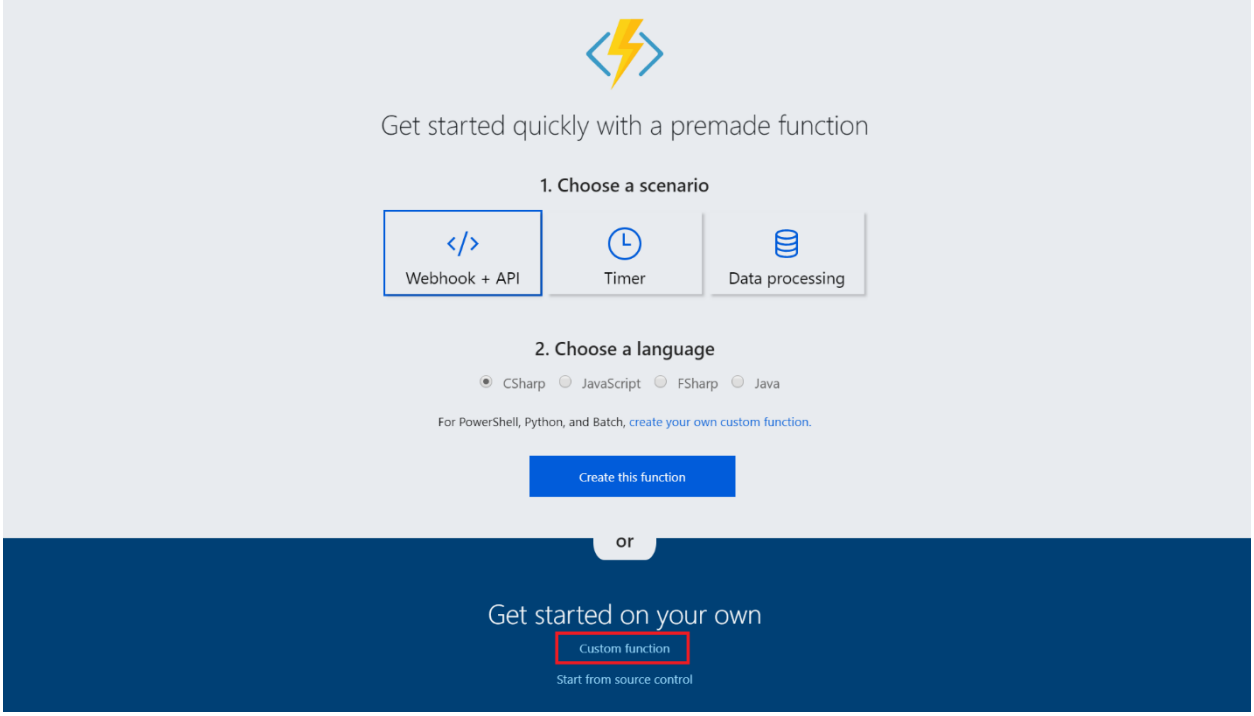
The Azure Function App is ready, and you have three containers to the storage account. The next step is to add an Azure Function.

Step 5 – Add the Azure Function

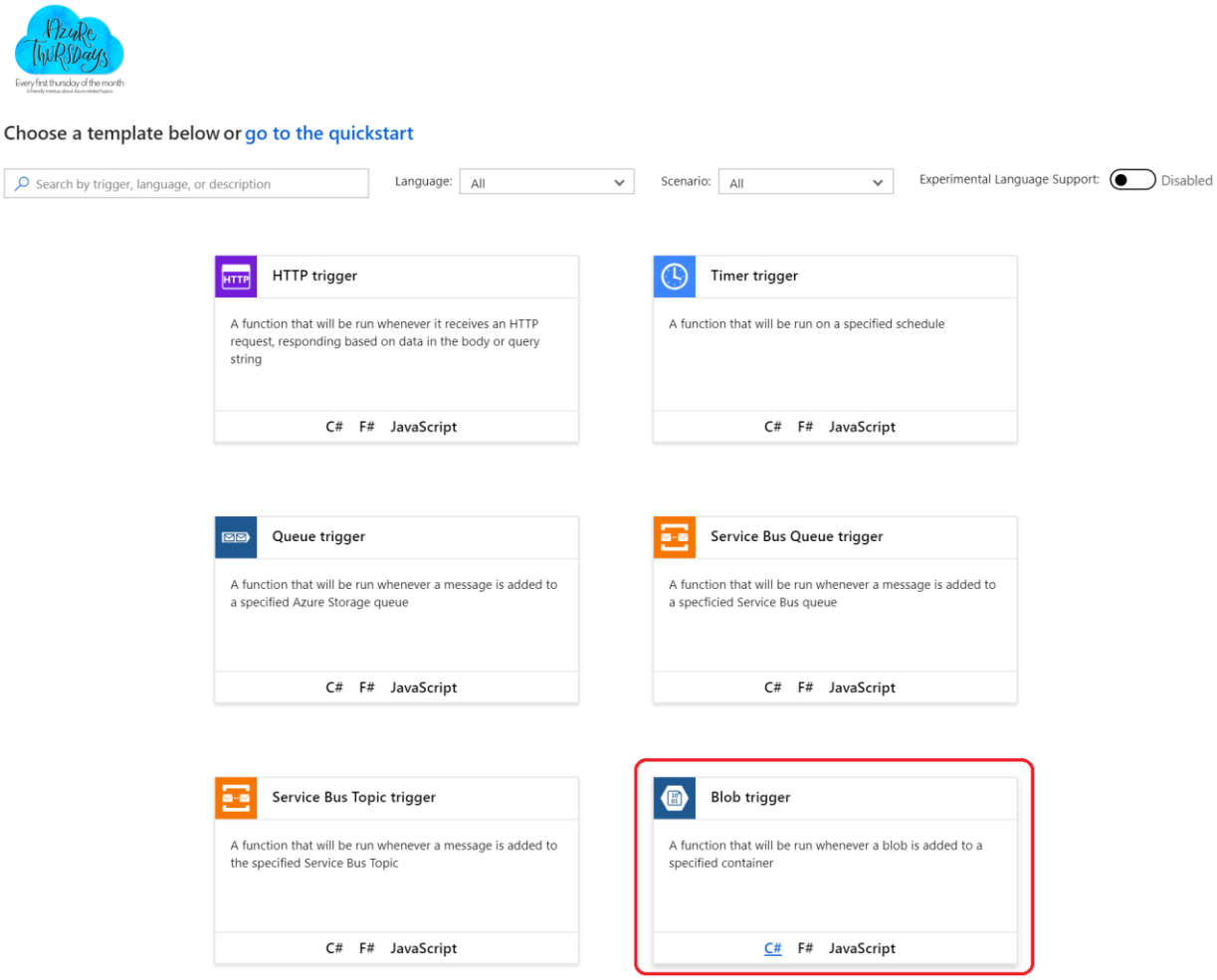
Once you have created an Azure Function App, you can add Azure Functions to it.

In this step, you will add a function to the Function App you created in step 2 and write C# code that uses the Computer Vision API to analyze images added to the "uploaded" container for adult or racy content.

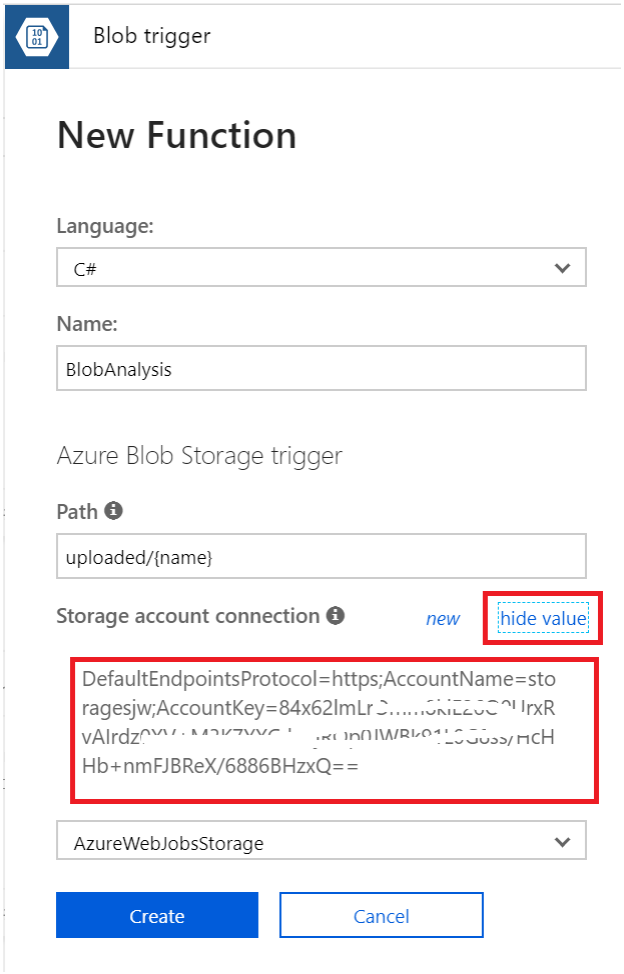
1. Click Azure Function App that you created in step 4.
2. Click the **+ sign** to the right of **Functions**.
3. Click custom function.



4. Select the Azure Blob Trigger Template (**C#**).



5. Specify a name like BlobAnalysis, check the storage account (click show value).
6. Change path to uploaded/{name} and click **Create**.



7. Replace the code shown in the code editor with the following statements:

   Note the endpoint (from Vision API): https://westeurope.api.cognitive.microsoft.com/vision/v2.0/

```
using Microsoft.WindowsAzure.Storage.Blob;
 using Microsoft.WindowsAzure.Storage;
 using System.Net.Http.Headers;
 using System.Configuration;

 public async static Task Run(Stream myBlob, string name, TraceWriter log)
 {
     log.Info($"Analyzing uploaded image {name} for adult content...");

     var array = await ToByteArrayAsync(myBlob);
     var result = await AnalyzeImageAsync(array, log);

     log.Info("Is Adult: " + result.adult.isAdultContent.ToString());
```

```
    log.Info("Adult Score: " + result.adult.adultScore.ToString());
    log.Info("Is Racy: " + result.adult.isRacyContent.ToString());
    log.Info("Racy Score: " + result.adult.racyScore.ToString());

    if (result.adult.isAdultContent || result.adult.isRacyContent)
    {
        // Copy blob to the "rejected" container
        StoreBlobWithMetadata(myBlob, "rejected", name, result, log);
    }
    else
    {
        // Copy blob to the "accepted" container
        StoreBlobWithMetadata(myBlob, "accepted", name, result, log);
    }
 }

 private async static Task<ImageAnalysisInfo> AnalyzeImageAsync(byte[] bytes, TraceWriter
log)
 {
    HttpClient client = new HttpClient();

    var key = ConfigurationManager.AppSettings["SubscriptionKey"];
    client.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key", key);

    HttpContent payload = new ByteArrayContent(bytes);
    payload.Headers.ContentType = new MediaTypeWithQualityHeaderValue("application/octet-
stream");

    var endpoint = ConfigurationManager.AppSettings["VisionEndpoint"];
    var results = await client.PostAsync(endpoint + "/analyze?visualFeatures=Adult",
payload);
    var result = await results.Content.ReadAsAsync<ImageAnalysisInfo>();
    return result;
 }

 // Writes a blob to a specified container and stores metadata with it
 private static void StoreBlobWithMetadata(Stream image, string containerName, string
blobName, ImageAnalysisInfo info, TraceWriter log)
 {
    log.Info($"Writing blob and metadata to {containerName} container...");

    var connection = ConfigurationManager.AppSettings["AzureWebJobsStorage"].ToString();
    var account = CloudStorageAccount.Parse(connection);
    var client = account.CreateCloudBlobClient();
    var container = client.GetContainerReference(containerName);
```

```
    try
    {
        var blob = container.GetBlockBlobReference(blobName);

        if (blob != null)
        {
            // Upload the blob
            blob.UploadFromStream(image);

            // Get the blob attributes
            blob.FetchAttributes();

            // Write the blob metadata
            blob.Metadata["isAdultContent"] = info.adult.isAdultContent.ToString();
            blob.Metadata["adultScore"] = info.adult.adultScore.ToString("P0").Replace("
","");
            blob.Metadata["isRacyContent"] = info.adult.isRacyContent.ToString();
            blob.Metadata["racyScore"] = info.adult.racyScore.ToString("P0").Replace("
","");

            // Save the blob metadata
            blob.SetMetadata();
        }
    }
    catch (Exception ex)
    {
        log.Info(ex.Message);
    }
 }

 // Converts a stream to a byte array
 private async static Task<byte[]> ToByteArrayAsync(Stream stream)
 {
    Int32 length = stream.Length > Int32.MaxValue ? Int32.MaxValue :
Convert.ToInt32(stream.Length);
    byte[] buffer = new Byte[length];
    await stream.ReadAsync(buffer, 0, length);
    stream.Position = 0;
    return buffer;
 }

 public class ImageAnalysisInfo
 {
    public Adult adult { get; set; }
```

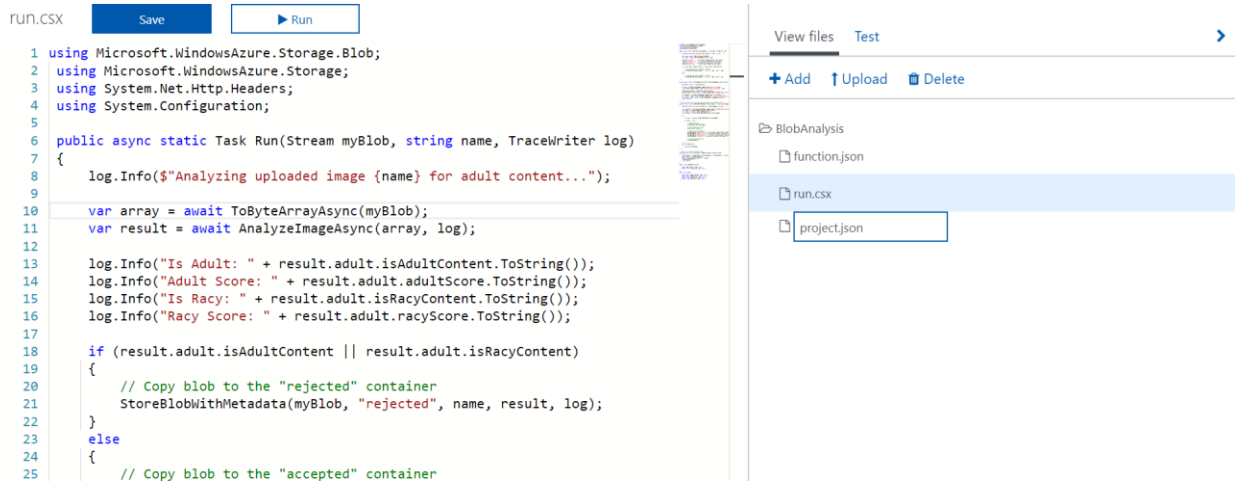
```
    public string requestId { get; set; }
}

public class Adult
{
    public bool isAdultContent { get; set; }
    public bool isRacyContent { get; set; }
    public float adultScore { get; set; }
    public float racyScore { get; set; }
}
```

Note check the Vision API endpoint (examine the swagger definition) : https://westeurope.dev.cognitive.microsoft.com/docs/services/5adf991815e1060e6355ad44/operations/56f91f2e778daf14a499e1fa

The endpoint should be: https://westeurope.api.cognitive.microsoft.com/vision/v2.0/analyze?visualFeatures=Adult

Run is the method called each time the function is executed. The Run method uses a helper method named AnalyzeImageAsync to pass each blob added to the "uploaded" container to the Computer Vision API for analysis. Then it calls a helper method named StoreBlobWithMetadata to create a copy of the blob in either the "accepted" container or the "rejected" container, depending on the scores returned by AnalyzeImageAsync.

8. Click the **Save** button at the top of the code editor to save your changes. Then click **View files**.
9. Click **+ Add** to add a new file and name the file **project.json**.

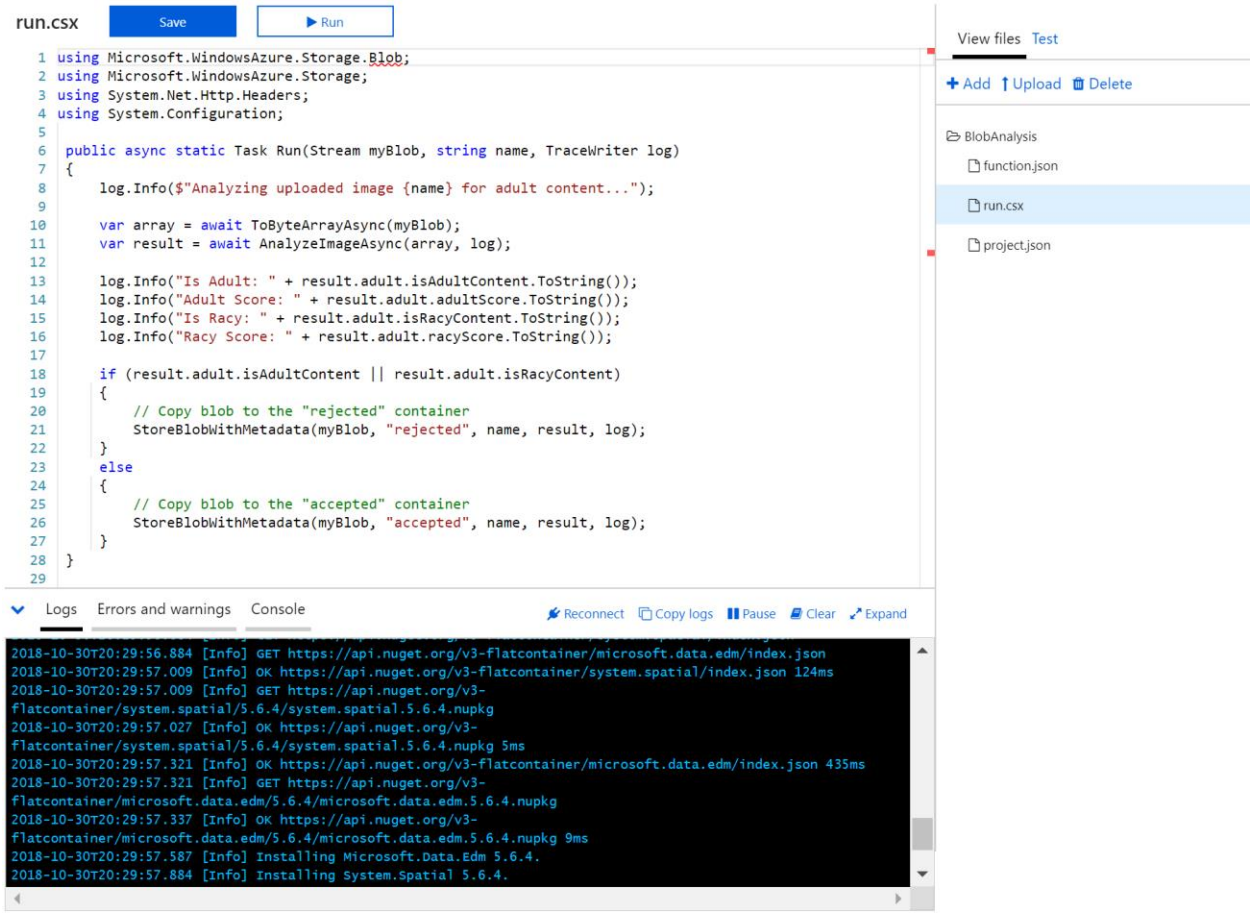


10. Add the following statement to the file:

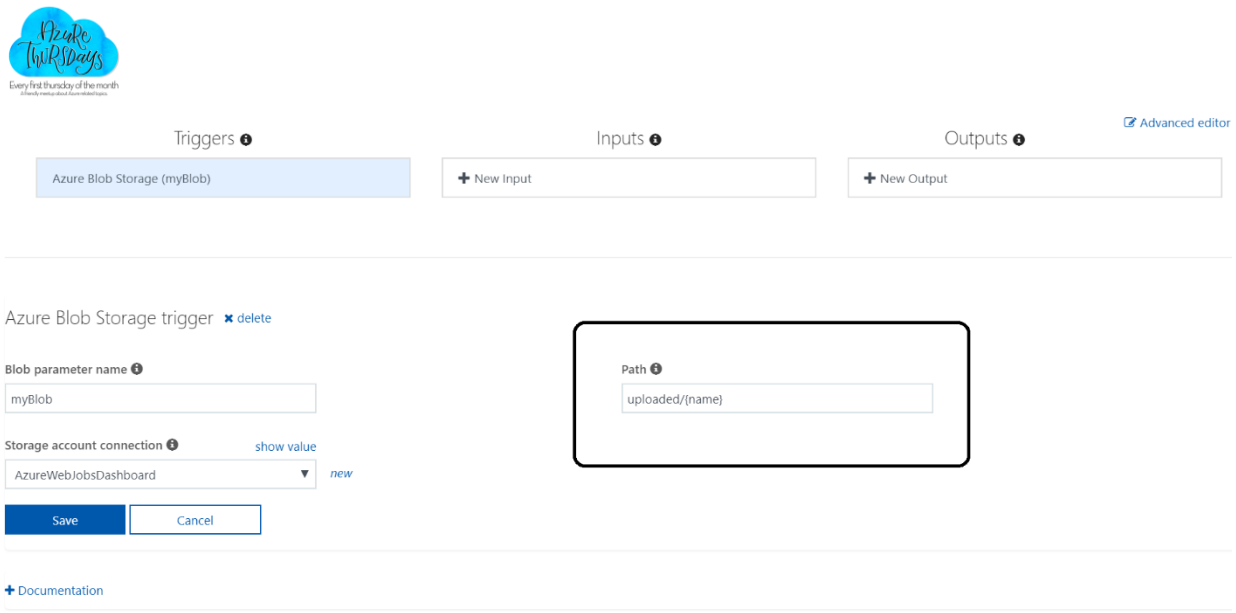
```
{
    "frameworks": {
```

```
        "net46": {
            "dependencies": {
                "WindowsAzure.Storage": "7.2.0"
            }
        }
    }
}
```

11. Return to run.csx.
12. Click Save, the NuGet packages will be restored.



13. Go to the Integrate tab of the function. Change the path to correct container name, i.e. uploaded if necessary.



You have now an example of an Azure Function written in C#, complete with a JSON project file containing information regarding project dependencies. The next step is to add an application setting that the Azure Function relies upon.

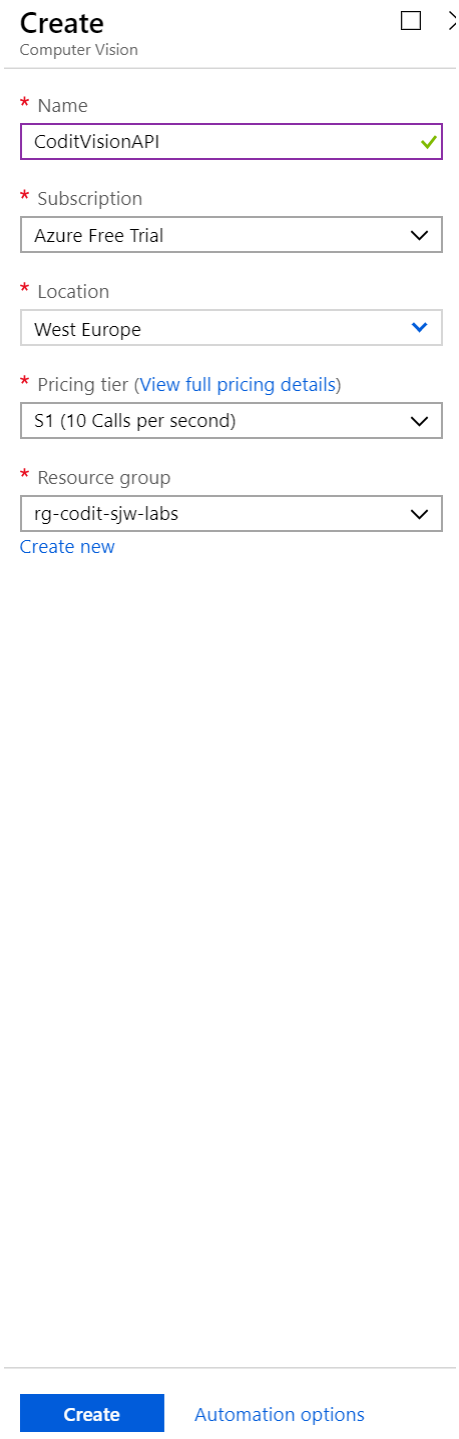
Step 6 – Create Cognitive Service API Instance

The Azure Function you created in step 5 loads a subscription key for the Microsoft Cognitive Services Computer Vision API from application settings. This key is required for your code to call the Computer Vision API and is transmitted in an HTTP header in each call. It also loads the base URL for the Computer Vision API (which varies by data center) from application settings.

Note that you can also try the quickstart for the Computer Vision API: https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/quickstarts-sdk/csharp-analyze-sdk

In this exercise, you will subscribe to the Computer Vision API, and then add an access key and a base URL to application settings.

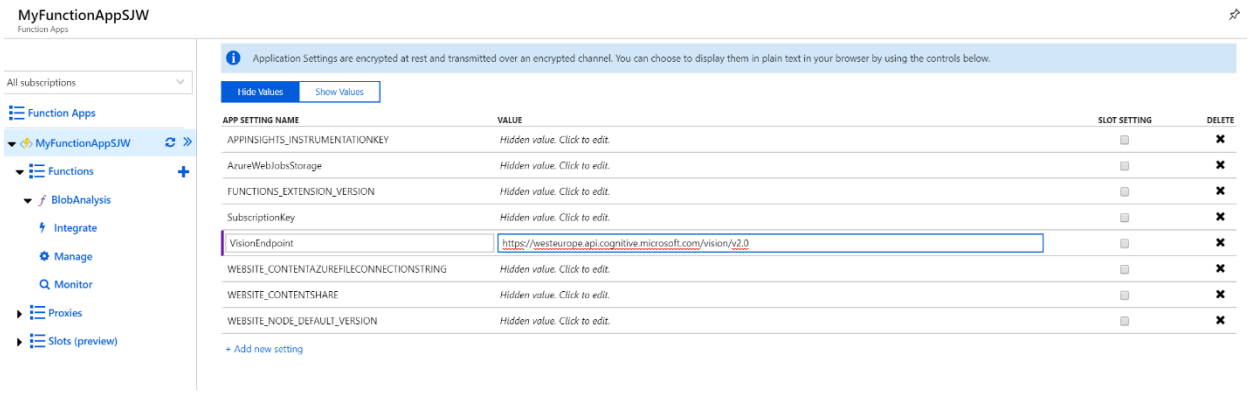
1. In the Azure Portal, go to your resource group, click **+ Create a resource**, in the search window enter **Computer Vision API**.
2. Select Computer Vision API and click create.
3. Name the service, select the correct location, tier, and resource group. Note that you need to choose **F0** is possible.



4. Return to the blade for you resource group and click the **Computer Vision API** subscription that you just created.
5. Copy the URL under **Endpoint** into your favourite text editor so you can quickly retrieve it in a moment. The complete endpoint should be: https://westeurope.api.cognitive.microsoft.com/vision/v2.0

Note see also Swagger documentation of the API:
https://westus.dev.cognitive.microsoft.com/docs/services/5adf991815e1060e6355ad44/operations/56f91f2e778daf14a499e1fa

6. Then click **Show access keys**.
7. Click the **Copy** button to the right of **KEY 1** to copy the access key to the clipboard.
8. Return to the **Function App** in the Azure Portal and click the app name in the ribbon on the left. Then click **Application settings**.
9. Scroll down to the "**Application settings**" section. Add a new app setting named "**Subscription Key**" (without quotation marks), and paste the subscription key that is on the clipboard into the Value box. Then add a setting named "**VisionEndpoint**" and set its value to the endpoint URL you saved in Step 5. Finish up by clicking **Save** at the top of the blade.



10. The app settings are now configured for your Azure Function.

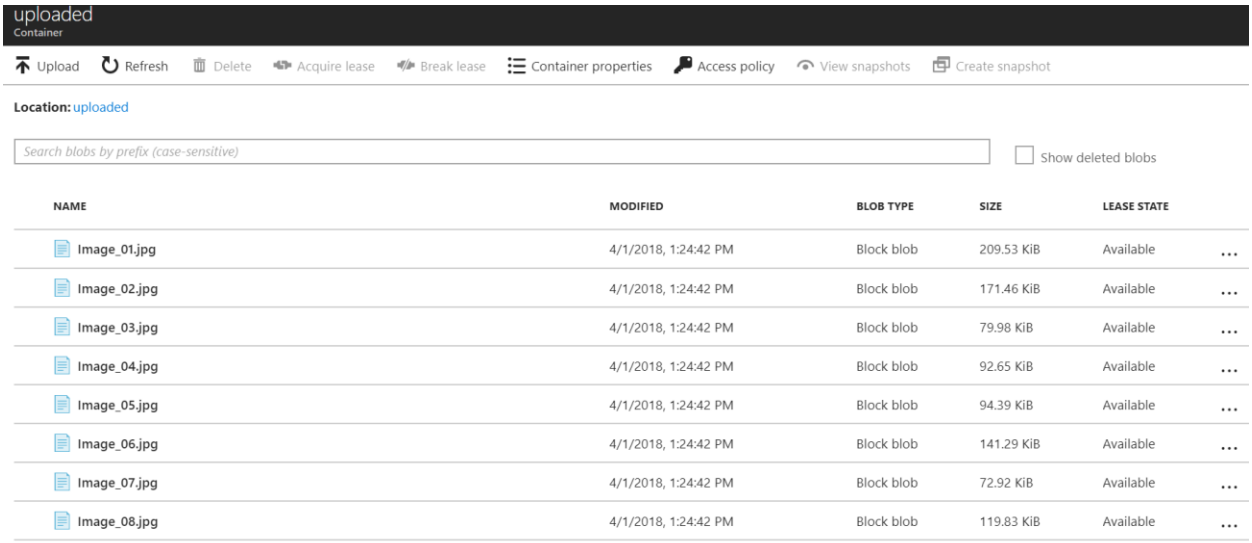
The work of writing and configuring the Azure Function is complete. Now comes the fun part: testing it out.

Step 7 - Test the Azure Function

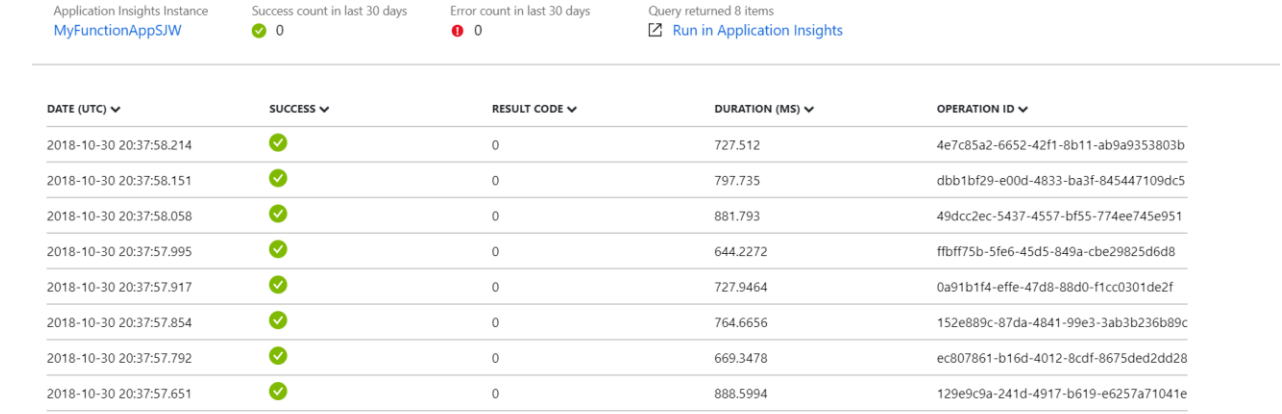
Your function is configured to listen for changes to the blob container named "uploaded" that you created in step 3. Each time an image appears in the container, the function executes and passes the image to the Computer Vision API for analysis.

To test the function, you upload images to the container. In this exercise, you will use the Azure Portal to upload images to the "uploaded" container and verify that copies of the images are placed in the "accepted" and "rejected" containers.

1. In the Azure Portal, go to the resource group created for your Function App. Then click the storage account that was created for it.
2. Click **Blobs** to view the contents of blob storage.
3. Click uploaded to open the "*uploaded*" container.
4. Click **Upload**.
5. Click the button with the folder icon to the right of the Files box. Select all of the files in this lab's "*Resources*" folder (or get the images from the images folder on GitHub or upload images yourself). Then click the **Upload** button to upload the files to the "uploaded" container.
6. Return to the blade for the "uploaded" container and verify that eight images are uploaded.



7. Close the blade for the "*uploaded*" container and open the "*accepted*" container.
8. Verify that the "*accepted*" container holds seven images. These are the images that were classified as neither adult nor racy by the Computer Vision API.
9. Close the blade for the "*accepted*" container and open the blade for the "*rejected*" container. Verify that the "*rejected*" container holds one image. This image was classified as adult or racy (or both) by the Computer Vision API.
10. Check the logs of the Function, by going to the function and check the logs.



Application Insights Instance: MyFunctionAppSJW | Success count in last 30 days: 0 | Error count in last 30 days: 0 | Query returned 8 items: Run in Application Insights

DATE (UTC)	SUCCESS	RESULT CODE	DURATION (MS)	OPERATION ID
2018-10-30 20:37:58.214	✓	0	727.512	4e7c85a2-6652-42f1-8b11-ab9a9353803b
2018-10-30 20:37:58.151	✓	0	797.735	dbb1bf29-e00d-4833-ba3f-845447109dc5
2018-10-30 20:37:58.058	✓	0	881.793	49dcc2ec-5437-4557-bf55-774ee745e951
2018-10-30 20:37:57.995	✓	0	644.2272	ffbff75b-5fe6-45d5-849a-cbe29825d6d8
2018-10-30 20:37:57.917	✓	0	727.9464	0a91b1f4-effe-47d8-88d0-f1cc0301de2f
2018-10-30 20:37:57.854	✓	0	764.6656	152e889c-87da-4841-99e3-3ab3b236b89c
2018-10-30 20:37:57.792	✓	0	669.3478	ec807861-b16d-4012-8cdf-8675ded2dd28
2018-10-30 20:37:57.651	✓	0	888.5994	129e9c9a-241d-4917-b619-e6257a71041e

The presence of seven images in the "*accepted*" container and one in the "*rejected*" container is proof that your Azure Function executed each time an image was uploaded to the "uploaded" container.

In this hands-on lab you learned how to:

- Create an Azure Function App
- Write an Azure Function that uses a blob trigger
- Add application settings to an Azure Function App
- Use Microsoft Cognitive Services to analyze images and store the results in blob metadata

This lab is just one example of how you can leverage Azure Functions to automate repetitive tasks. Experiment with other Azure Function templates to learn more about Azure Functions and to identify additional ways in which they can aid your research or business.