

Azure Development

Lab 3

Introduction of Azure Functions

DISCLSIMER

© 2025 Microsoft Corporation. All rights reserved. Microsoft, Windows and other product names are or may be registered trademarks and/or trademarks in the U.S. and/or other countries.

The information herein is for informational purposes only and represents the current view of Microsoft Corporation or any Microsoft Group affiliate as of the date of this presentation. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information provided after the date of this presentation.

MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS PRESENTATION.

CONFIDENTIALITY

© 2025 Microsoft Corporation. All rights reserved. Any use or distribution of these materials without express authorization of Microsoft Corp. is strictly prohibited.

Contents

Lab 1 – Introduction of Azure Functions 4

Exercise 1- Set up Working Environment for Functions (60min)5

Module 1 – Create App Service Plan in Azure Portal 6

Module 2 – Create Azure Function in Azure Portal 6

Module 3 – Create Blob Containers 7

Module 4 – Set up Prerequisites for Visual Studio 8

Module 5 – Create Function Project in Visual Studio9

Module 6 – Convert In-process Project to Isolated Worker Model Project 11

Module 7 – Add Program.cs 12

Module 8 – Convert Your Function Code 13

Module 9 – Validate Configuration File 14

Module 10 – Run and Test Locally 15

Summary 16

Exercise 2- Deploy Your Function to Azure (15min) 18

Module 1 – Understand Your Connection Strings in Azure 18

Module 2 – Deploy Your Function to Azure 18

Module 3 – Configure Application Settings in Azure 18

Module 4 – Test Your Function 18

Summary 19

Exercise 3- Implement Function (15min) 20

Module 1 – Implement Classify Image Function 20

Module 2 – Create Azure AI Vision Service 24

Module 4 – Add Application Settings in VS 24

Module 5 – Test Your Function Locally 25

Summary 25

Exercise 4- Deploy Your Function (15min) 26

Module 1 – Publish Your Function 26

Module 2 – Add App Settings for Computer Vision Service 26

Module 3 – Test Your Function 26

Summary 26

Exercise 5 (Optional)- Manage Secrets (min) 27

Module 1 – Publish Your Function 27

Lab 1 – Introduction of Azure Functions

Objective(s)	<ul style="list-style-type: none">• To•
Duration of Lab	<ul style="list-style-type: none">• 2h
Prerequisite(s)	<ul style="list-style-type: none">• Azure Subscription and Resource Group• Contributor Role in a selected Resource Group• Access to the Internet
Tool(s)	<ul style="list-style-type: none">• Azure Portal
Exercises	1.
Subscription	[Selected Subscription]
Resource Group	[Selected RG]
Navigation	Throughout this Lab, we will open and use several Browser tabs for easy access. Until the end of the Lab, keep your Browser tabs open.
References	<ul style="list-style-type: none">• Azure Hands on Labs - Serverless with Azure Functions

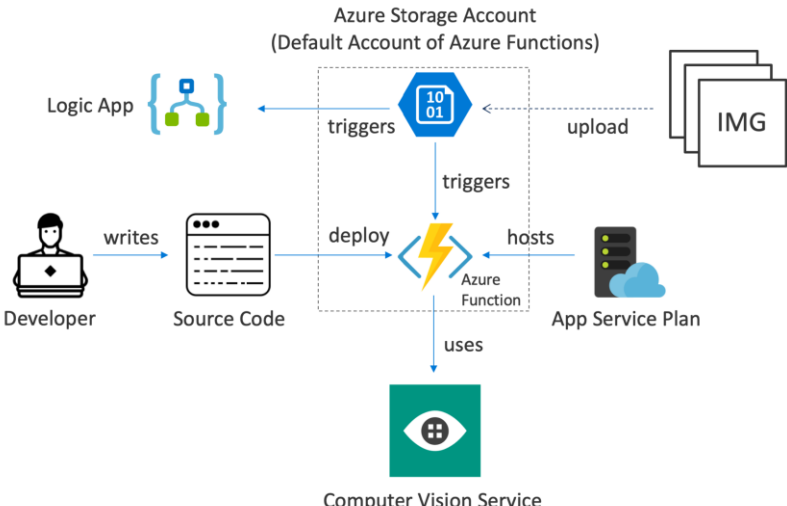
Naming Convention for Labs

For completing various labs during the workshops, we will use this naming convention. It is slightly different from Microsoft online guidance ([Define your naming convention - Cloud Adoption Framework | Microsoft Learn](#)).

The naming convention below is designed to group your Azure resources together for easy access.

[you name/initials]-[short name for Azure service]-[service description]

Exercise 1- Set up Working Environment for Functions (60min)

<p>Lab Scenario</p>	<p>You will create an Azure Function that monitors a blob container in Azure Storage for new images, and then performs automated analysis of the images using the Microsoft Cognitive Services Computer Vision API.</p> <p>Specifically, The Azure Function will analyse each image that is uploaded to the container for adult or racy content and create a copy of the image in another container.</p> <p>Images that contain adult or racy content will be copied to one container, and images that do not contain adult or racy content will be copied to another.</p> <p>In addition, the scores returned by the Computer Vision API will be stored in blob metadata.</p> <p>Finally, you will create an Azure Logic App to create a workflow to notify about rejected image</p> <p>This lab is available from the online hands-on Azure Hands on Labs - Serverless with Azure Functions.</p> <p>We will follow the flows of lab exercises. However, there are discrepancies in the instructions regarding how Azure resources can be created in the current state.</p>
<p>Architecture</p>	 <pre> graph TD Dev[Developer] -- writes --> SC[Source Code] SC -- deploy --> AF[Azure Function] AF -- triggers --> LA[Logic App] AF -- triggers --> AS[Azure Storage Account] AS -- upload --> IMG[IMG] AF -- uses --> CVS[Computer Vision Service] AF -- hosts --> ASP[App Service Plan] </pre>
<p>Objectives</p>	<ul style="list-style-type: none"> • 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 • Use Azure Logic Apps to build serverless workflows
<p>Tools</p>	<ul style="list-style-type: none"> • Visual Studio 2022 • .NET 8 SDK • Azure Portal
<p>Topics</p>	<p>In this exercise, we will cover the following topics.</p> <ul style="list-style-type: none"> • Create App Service Plan in Azure Portal • Create Azure Function in Azure Portal • Create Blob Containers • Set up Prerequisites for Visual Studio • Create Function Project in Visual Studio

	<ul style="list-style-type: none"> • Convert In-process Project to Isolated Worker Model Project • Add Program.cs • Convert Your Function Code • Validate Configuration File • Run and Test Locally
Duration	<ul style="list-style-type: none"> • 60 min
Tool(s)	<ul style="list-style-type: none"> • Azure portal • Visual Studio 2022 • Azure Storage Explorer • Azurite • Command Prompt
Subscription	[selected subscription]
Resource Group	[selected RG]
Reference(s)	<ul style="list-style-type: none"> • Guide for running C# Azure Functions in an isolated worker process Microsoft Learn

Module 1 – Create App Service Plan in Azure Portal

- Follow the instructions on [Azure Hands on Labs - Serverless with Azure Functions](#) – Exercise 1
 - App Service Plan name: [your short name]-vision-analysis-plan
 - Region: Australia East
- Complete the session

Module 2 – Create Azure Function in Azure Portal

- Create a new resource for **Function App**
- In the very beginning of provisioning Function, you will see this prompt first

Create Function App

Select a hosting option

These options determine how your app scales, resources available per instance, and pricing. [Learn more about Functions hosting options](#)

Hosting plans	Flex Consumption	Consumption	Functions Premium	App Service	Container Apps environment
	Get high scalability with compute choices, virtual networking, and pay-as-you-go billing.	Pay for compute resources when your functions are running (pay-as-you-go).	Deploy multiple function apps on the same plan with event-driven scaling.	Run web apps and function apps on the same plan with more compute choices and pay for the instances of the plan.	Host function apps with other containerized microservices and pay for compute capacity.
Scale to zero	✓	✓	-	-	✓
Scale behavior	Fast event-driven	Event-driven	Event-driven	Metrics based	Event-driven with KEDA
Virtual networking	✓	-	✓	✓	✓
Dedicated compute and prevent cold start	Optional with Always Ready	-	Minimum of 1 instance required	Minimum of 1 instance required	Optional with minimum replicas
Max scale out (instances)	1000	200	100	30	300

-
-
- Note: 1. Azure portal and provisioning experience have made the hosting plan selection more explicit and restrictive in some scenarios
-
- Select App Service
- Enter the following details

Item	Description
Subscription	[your subscription]
Resource Group	[your RG]
Function App name	[your short name]-fn-vision-analysis
Try secure unique default hostname	(unchecked)
Code or Container	Code
Operating System	Windows
Runtime stack	.NET
Version	9 (LTS) isolated worker model
Region	Australia East
Windows Plan	[your App Service created for this lab] – S1
Storage	
Storage account	(you can use a default name of storage account. Otherwise, create new for yourself. It will make it easier for you later on to identify throughout the lab) [your short name]stavisionanalysis
Monitoring	
Enable Application insights	Enable
Application Insights	(you can use a default name of Application Insight. Otherwise, create new for yourself) [your short name]-appisgt-vision-analysis (default Workspace)
Authentication	
Authentication Type	Managed Identity (use default)

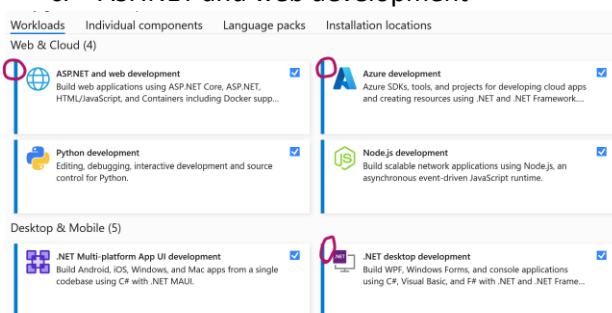
5. Review + create and Create

Module 3 – Create Blob Containers

1. Select your newly created Storage Account for the lab
2. Select Containers under the Data storage section
3. Follow the instructions on [Azure Hands on Labs - Serverless with Azure Functions](#) only for **Task 3:**
Upload the test data
4. You have created 3 containers in your Blob Storage – ***uploaded, accepted, rejected***
5. At this point, you are not allowed to add files (blobs) as you have not assigned the Data Plane level permissions. If you try to upload a file (blob), you will get the unauthorised error
6. Select Access Control (IAM) -> Add role assignment
7. Type “**blob**” in the search textbox and find the *Storage Blob Data Contributor* role
8. Click **Next** and add yourself

Module 4 – Set up Prerequisites for Visual Studio

1. Prerequisites for working with Aure Functions isolated worker model in Visual Studio, make sure you have the following tools installed and set up:
 - a. Visual Studio 2022 (v17.8 or later) with:
 - i. Azure Development workload
 - ii. .NET 8 SDK
 - b. Azure Functions Core Tools v4
 - c. Azure Storage Emulator or access to Azure Storage Account
2. In VS, open Developer Command Prompt window
3. Check the .NET version by typing ***dotnet --version***
4. Make sure the following workloads are installed for VS. Go to **Tools -> Get Tools and Features...**
 - a. Azure Development
 - b. .NET Desktop Development
 - c. ASP.NET and web development



- 5.
- 6.
5. Install Azure Functions Core Tools (v4). This is needed to run and test Azure Functions locally
6. Open PowerShell or Command Prompt (requires Node.js to be installed):

```
npm install -g azure-functions-core-tools@4 --unsafe-perm true
```

7. Verify:

```
func --version
```

8. You should see something like 4.x.x
9. Install *Azure Storage Emulator* or Use *Azure Storage* for triggers like Blob, Queue etc....
 - a. Use Azure Storage
 - i. Create a Storage Account in Azure Portal
 - ii. Copy the connection string from access Keys
 - b. Use Azurite (local emulator)
 - i. Install via npm - `npm install -g azurite`
 - ii. Run it: ***azurite***
10. Install Azure Storage Explorer (optional, but helpful)
 - a. If you want to connect to Azure Storage and work locally, it will be helpful to download *Azure Storage Explorer* and install it

Module 5 – Create Function Project in Visual Studio



Azure Functions Process Models

Azure Functions has 2 types of **process models** – **in-process model** (old way) and **isolated worker model** (recommended, .NET6+ and .NET8).

There are key differences between 2 models that impacts how you need to code and develop. Make sure that you understand key differences and use isolated worker model.

Note: Visual Studio is still defaulted to using in-process model template for Azure Functions. Make sure you create

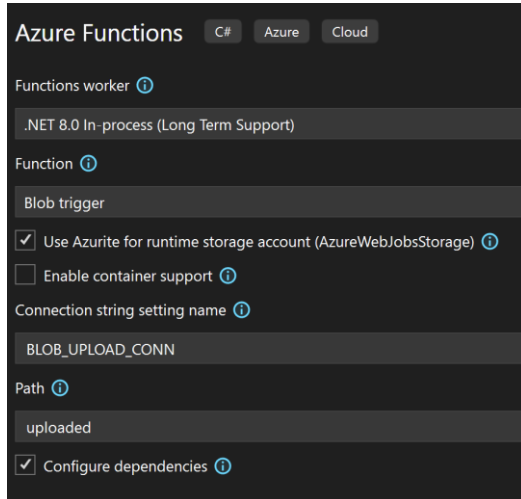
	In-process model (old way)	Isolated worker model (recommended. .NET6+ and more)
Characteristics	<ul style="list-style-type: none"> Runs inside the Azure Functions host process. Uses the same runtime as the host (<i>Microsoft.NET.SDK.Functions</i>). Tightly coupled with Azure Functions SDK. 	<ul style="list-style-type: none"> Runs in a separate process from the Azure Functions host. Uses <i>Microsoft.Azure.Functions.Worker SDK</i>. Full control over the .NET runtime and dependencies.
VS Project Structure	<ul style="list-style-type: none"> No <i>Program.cs</i>. Functions are decorated with <i>[FunctionName]</i> and use bindings directly. Logging via <i>ILogger</i> injected into the function method. 	<ul style="list-style-type: none"> Requires a <i>Program.cs</i> to configure the host. Functions use <i>FunctionContext</i> to access services like logging. More explicit setup for DI, configuration, and logging.
Pros	<ul style="list-style-type: none"> Simpler for small apps. Less boilerplate. 	<ul style="list-style-type: none"> Full control over .NET runtime (e.g., .NET 8). Better support for modern .NET features (e.g., minimal APIs, DI). Easier unit testing and separation of concerns.
Cons	<ul style="list-style-type: none"> Limited to .NET versions supported by the host. Less flexibility for DI (Dependency Injection), middleware, and custom configuration. Harder to test and debug in isolation. 	<ul style="list-style-type: none"> Slightly more complex setup. Some bindings may require additional packages (e.g., Blob, Queue).
Example	<pre>[FunctionName("MyFunction")] public void Run([TimerTrigger("0 */5 * * *")] TimerInfo myTimer, ILogger log) { log.LogInformation("Function executed"); }</pre>	<pre>[Function("MyFunction")] public void Run([TimerTrigger("0 */5 * * *")] TimerInfo myTimer, FunctionContext context) { var logger = context.GetLogger("MyFunction"); logger.LogInformation("Function executed"); }</pre>

7. IMPORTANT

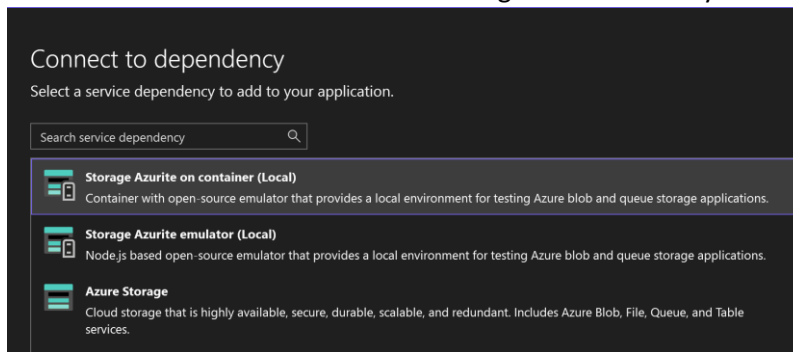
- Even after you have installed all prerequisites and tools, Visual Studio might not reliably include .NET 8 Isolated Worker template for Azure Functions out of the box.

9. This is a known issue and has been widely reported by developers.
- 10.
11. You can create the project using Azure Functions Core Tools CLI. We will not discuss the details for this lab

12. Open **Visual Studio** (for this exercise, Visual Studio 2022)
13. Create a new **Project** using **Azure Functions** template



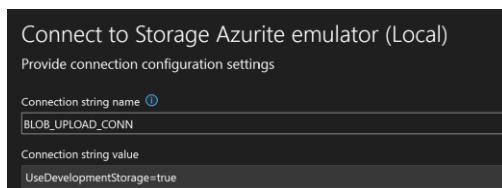
- 14.
15. Click **Next** and select **Emulator** for running Functions locally



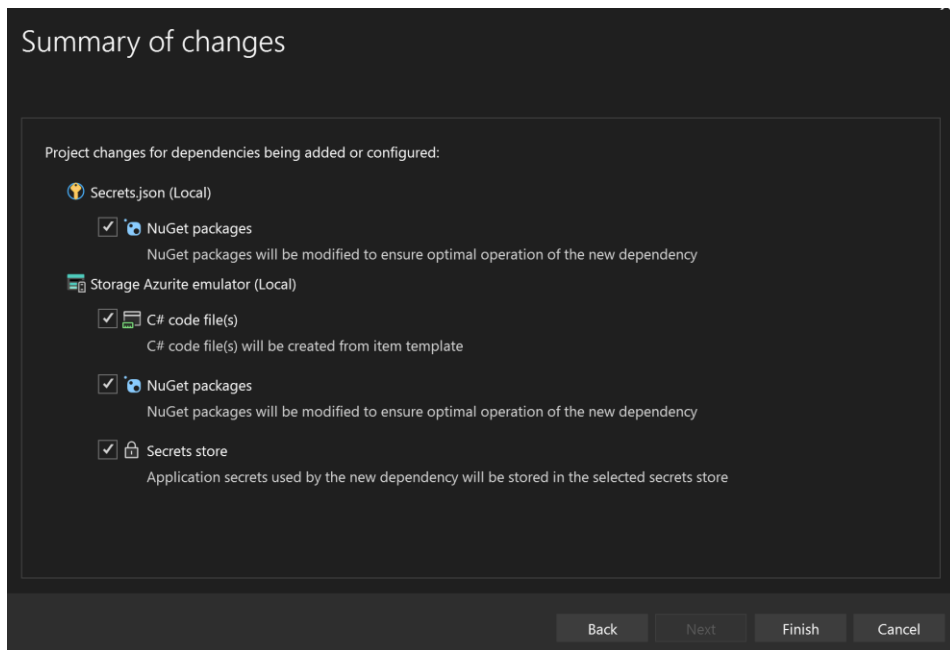
- 16.

Option	Description	Use Case
Azurite on container (local)	Runs Azurite (Azure Storage emulator) inside a Docker container. Requires Docker Desktop.	Best for isolated local dev environments using containers.
Azurite emulator (local)	Runs Azurite as a local Node.js process (no Docker).	Lightweight local dev without Docker.
Azure Storage	Uses a real Azure Storage account in the cloud.	For integration testing or production-like environments.

17. Click **Next**



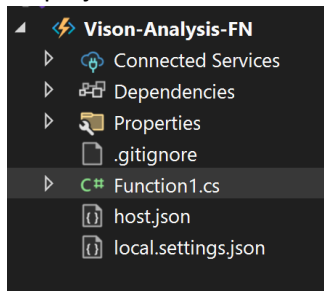
- 18.
19. Click **Next** and **Finish**



20.

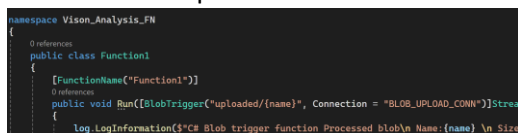
21.

22. VS project looks like this



23.

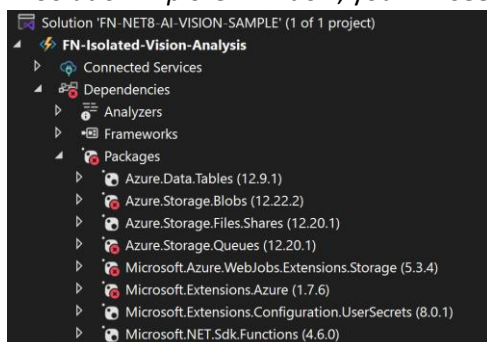
24. The default template function is added for you



25.

Module 6 – Convert In-process Project to Isolated Worker Model Project

1. In *Solution Explorer* window, you will see several error icons for your packages



26.

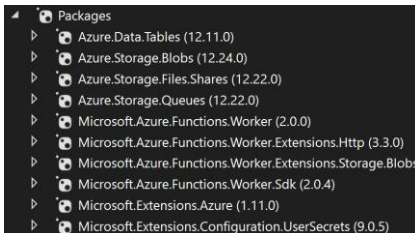
2. Remove Microsoft.NET.Sdk.Functions (4.6.0) and Microsoft.Azure.WebJobs.Extensions.* (those packages are for in-process Functions)

3. Open **Manage NuGet Packages...** (from Tools-> NuGet Package Manager or the context menu from Solution Explorer)


- Search and install the following packages one by one:

Package Name	Notes
Microsoft.Azure.Functions.Worker	Core SDK for isolated model
Microsoft.Azure.Functions.Worker.Sdk	Required for build-time support (set as Analyzer)
Microsoft.Azure.Functions.Worker.Extensions.Http	Enables HTTP trigger support
Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs	Extensions for Blob triggers, etc...

- Save changes



Package Warning

The yellow icon  just means the package was built for an older version of .NET, but it still works perfectly with our current version. It's like using a charger made for an older phone model—it still fits and works fine.

- Run **Clean** and **Rebuild** processes to make sure there is no errors

Module 7 – Add Program.cs

- Create a new file in the root of your project named **Program.cs** and add the following code:

```
using Microsoft.Extensions.Hosting;

var host = new HostBuilder()
    .ConfigureFunctionsWorkerDefaults()
    .Build();

host.Run();
```



New Way of Working for .NET 8 or later

In the **.NET isolated worker model** for Azure Functions, the *Program.cs* file **does not require a class**. It uses a **top-level statement** approach, which is a feature introduced in C# 9 and fully supported in .NET 6 and later.

In traditional .NET apps, you'd see something like:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }
}
```

But in modern .NET (including Azure Functions isolated worker), you can write:

```
using Microsoft.Extensions.Hosting;

var host = new HostBuilder()
    .ConfigureFunctionsWorkerDefaults()
    .Build();

host.Run();
```

(No *class* declaration, no *namespace* needed)

Module 8 – Convert Your Function Code

1. Before (In-Process):

```
[FunctionName("ClassifyImage")]
public void Run(
    [BlobTrigger("images/{name}")] Stream image,
    string name,
    ILogger log)
{
    log.LogInformation($"Processing image: {name}");
}
```

27.

2. After (Isolated Worker):

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Logging;

namespace FN_Isolated_Vision_Analysis
{
    1 reference
    public class ClassifyImage
    {
        [Function("ClassifyImage")]
        1 reference
        public async Task RunAsync(
            [BlobTrigger("uploaded/{name}", Connection = "BLOB_CONN")] Stream myBlob,
            string name,
            FunctionContext context)
        {
            var logger = context.GetLogger("ClassifyImage");
            logger.LogInformation($"C# Blob trigger function Processed blob\n Name:{name} \n Size: {myBlob.Length} Bytes");

            // Optional: await any async processing here
            await Task.CompletedTask;
        }
    }
}
```

28.

3. You will modify the highlighted codes

4. If you class name is still *Function1.cs*, rename it to ***ClassifyImage.cs***

5. Rename *Function* within the class to ***ClassifyImage***

6. Remove *ILogger* and *HttpRequest* references. You no longer use *ILogger* directly in the method signature. Instead, you use ***FunctionContext*** and retrieve the logger from it

7. Get Logger instance from *FunctionContext*

8. Make your function ***async***

9. Key changes explained here:

In-Process	Isolated Worker
[FunctionName]	[Function]
ILogger log	FunctionContext context
Direct log.LogInformation(...)	Context.GetLogger("...").LogInformation(...)
Sync method	Prefer async Task for extensibility

Module 9 – Validate Configuration File

1. In VS, open local.settings.json file
2. Update **local.settings.json** file like this below

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "BLOB_CONN": "UseDevelopmentStorage=true",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated"
  }
}
```

29. IMPORTANT

30. The Azure Functions runtime uses **"AzureWebJobsStorage"** to manage internal operations, such as:

- Logging
- Checkpointing
- Timer triggers
- Scale control (in Azure)

Even if your function bindings (like Blob or Queue) use a **custom connection string**, the host still expects **"AzureWebJobsStorage"** to be present.



New Way of Working for .NET 8 or later

It is actually a best practice in many cases to separate azureWebJobsStorage and create a custom connection for your Blob.

- Separation of Concerns
- Security Access control
- Performance Optimisation
- Cross-Region or Multi-Cloud Scenarios
- Local Development vs. Production

Locally, you might use:

```
"AzureWebJobsStorage": "UseDevelopmentStorage=true",
"BLOB_CONN": "UseDevelopmentStorage=true"
```

In production:

```
"AzureWebJobsStorage": "<runtime-storage-connection>",
"BLOB_CONN": "<app-data-storage-connection>"
```

3. Update your **launchSettings.json**. This file is mainly used for debugging and launching behaviour in VS (e.g. ports, browser launch, and minimal environment setup)
4. In *Solution Explorer*, expand the **Properties** folder of your project
5. Double-click launchSettings.json file
6. Replace the existing with the following

```
{
  "profiles": {
    "FN_Isolated_Vision_Analysis": {
      "commandName": "Project",
      "commandLineArgs": "--port 7295",
      "launchBrowser": false,
      "environmentVariables": {
        "AzureWebJobsStorage": "UseDevelopmentStorage=true",
        "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated"
      }
    }
  }
}
```

```
{
  "profiles": {
    "FN_Isolated_Vision_Analysis": {
      "commandName": "Project",
      "commandLineArgs": "--port 7295",
      "launchBrowser": false,
      "environmentVariables": {
        "AzureWebJobsStorage": "UseDevelopmentStorage=true",
        "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated"
      }
    }
  }
}
```

7. Save changes

Module 10 – Run and Test Locally

1. Start **Azurite**, or your storage emulator is running
2. Open **Command Prompt** in Administrator mode and run **azurite**



Azurite and running Functions locally

Azurite uses the default ports for Blob, Queue and Table. Some functionality within Azurite and/or Azure Functions Tools are set to those default ports, even if you want to run those using different ports (e.g. `azurite --blobHost 127.0.0.1 --blobPort 10010 --queueHost 127.0.0.1 --queuePort 10011 --tableHost 127.0.0.1 --tablePort 10000`)

3. If you get an error like this, it means that Azurite is trying to **bind a port (usually 10000 for Blob service) that is already in use** – likely by another instance of Azurite or another application)

```
Azurite Blob service is starting at http://127.0.0.1:10000
Exit due to unhandled error: listen EADDRINUSE: address already in use 127.0.0.1:10000
```

- 31.
4. Open Command Prompt or PowerShell
5. Run this command to find what's using port 10000:

```
netstat -aon | findstr :10000
```

6. If another process is already using the same port, find out what process is using by running this command. Replace <PID> with the actual process ID found:

```
tasklist /FI "PID eq <PID>"
```

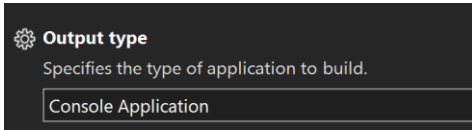
7. Make sure the Connection String for Storage Account is all the same in *local.settings.json* and *launch.json*

```
"AzureWebJobsStorage": "UseDevelopmentStorage=true"
```

And

```
"BLOB_CONN": "UseDevelopmentStorage=true"
```

8. Run your Function locally (i.e. F5)
9. If you get an error like *"Program using top-level statements must be an executable"*, your project configuration is wrong
10. Check your Function project is made as **Executable** in VS
11. Right-click the project in Solution Explorer (e.g., FN-Isolated-Vision-Analysis), and Click **Properties**
12. In the left-hand menu, select **Application**. Look for the **Output type** dropdown.
13. Change it from Class Library to **Console Application**



- 32.
14. Rebuild the solution (make sure Azure Functions Core Tools v4.0.5535 or later)

```
npm i -g azure-functions-core-tools@4 --unsafe-perm true
```

15. Open **Command Prompt** in Administrator mode
16. Run Azurite - *azurite*
17. In VS, run the solution (i.e. **F5**). Or run the command *'func start'* in *Developer Command Prompt* Window
18. Upload a file using Azure Storage Explorer or azcopy function in Command Prompt

Overall steps to run Functions locally using Azure Functions Core tool (v4) and Azurite are:

1. Open *Command Prompt* and run **azurite**
2. Open *Azure Storage Explorer* and connect to azurite local server (emulator connection)
3. Create a new container named *"uploaded"*
4. In VS, run the command, **func start**, in *Developer Command Prompt*
5. In *Storage Explorer*, upload a file

Summary



ACHIEVEMENTS

After you have completed the exercise, you are now able to:

- ✓ Prepare local environment for working with Azure Functions in Visual Studio
- ✓ Understand various configuration settings used and required (e.g. *local.settings.json*, *launchSettings.json*)
- ✓ Understand emulator using Azurite
- ✓ Run Azurite emulator

- ✓ Set up Azure Storage Explorer
- ✓ Fun Function locally from Visual Studio

Exercise 2- Deploy Your Function to Azure (15min)

Topics	<p>In this exercise, we will cover the following topics.</p> <ul style="list-style-type: none">• Understand Your Connection Strings• Deploy Your Function to Azure• Configure App Settings• Test Your Function
Duration	<ul style="list-style-type: none">• 15 min
Tool(s)	<ul style="list-style-type: none">• Azure portal
Subscription	[selected subscription]
Resource Group	[selected RG]

Module 1 – Understand Your Connection Strings in Azure

1. *local.settings.json* is for local development only. Its values are not uploaded to Azure
2. After deployment, you must set your connection strings (**AzureWebJobsStorage**, **BLOB_CONN**) in the Azure Portal
3. The connection string for your Azure Storage account (not **UseDevelopmentStorage=true**)

Module 2 – Deploy Your Function to Azure

1. **Right-click** your project in *Solution Explorer*
2. Select **Publish**
3. Choose Azure > Azure Function App (Windows) (or Linux, as appropriate)
4. Click **Next** and follow the prompts to create/select a Function App in your Azure subscription
5. Click **Finish** and then **Publish**

Module 3 – Configure Application Settings in Azure

1. In *Azure portal*, go to your deployed **Function App**
2. In the left menu, select **Environment variables** under *Settings*
3. Delete the extra **AzureWebJobsStorage_*** entries
4. Under **Application settings**, add or update the following keys:
 - a. **AzureWebJobsStorage** - set to your Azure Storage account connection string
 - b. **FUNCTIONS_WORKER_RUNTIME** - should already be set to *dotnet-isolated*
 - c. **BLOB_CONN** - set to your Azure Storage account connection string (or another if you use a different storage account).
5. Click Save

Module 4 – Test Your Function

1. In your Function App, open **Log Stream** (e.g. type 'log stream' in the search textbox)

2. Once the Log Stream is connected to App Insights Logs, open another browser and go to your Storage Account in Azure portal
3. Select **“uploaded”** container and upload one of sample images provided
4. Go back to your Function App, check the logs
5. You will see the successful Blob trigger

```

Connected!
2025-06-01T07:46:43Z [Information] C# Blob trigger function Processed blob
Name: Image_01.jpg
Size: 214003 Bytes
2025-06-01T07:46:43Z [Information] Executed 'Functions.ClassifyImage'
(Succeeded, Id=a7b08336-24c1-48b2-be55-0fa3e23c2a3, Duration=2647ms)
2025-06-01T07:46:43Z [Information] HttpOptions
{
  "DynamicThrottlesEnabled": false,
  "PublicChunkedRequestBinding": false,
  "MaxConcurrentRequests": -1,
  "MaxOutstandingRequests": -1,
  "RoutePrefix": "api"
}
2025-06-01T07:46:43Z [Information] BlobsOptions
{
  "MaxDegreeOfParallelism": 8,
  "PoisonBlobThreshold": 5
}
2025-06-01T07:46:43Z [Information] ConcurrencyOptions
{
  "DynamicConcurrencyEnabled": false,
  "MaximumFunctionConcurrency": 500,
  "CPUThreshold": 8.8,
  "SnapshotPersistenceEnabled": true
}

```

- 33.
6. Move away from Log Stream. This will kill the real-time logging screen
7. Remove the uploaded file from Storage Blob for the next exercises

Summary



ACHIEVEMENTS

After you have completed the exercise, you are now able to:

- ✓ Publish your Function App in Azure
- ✓ Test your Function App using Log Stream
- ✓ Configure correct application settings for your Function App in Azure

Exercise 3- Implement Function (15min)

Topics	In this exercise, we will cover the following topics. <ul style="list-style-type: none"> • Implement ClassifyImage.cs • Create Azure Computer Vision Service • Add Application Settings • Test Your Function Locally
Duration	<ul style="list-style-type: none"> • 15 min
Tool(s)	<ul style="list-style-type: none"> • Azure portal • Visual Studio 2022
Subscription	[selected subscription]
Resource Group	[selected RG]

Module 1 – Implement Classify Image Function

1. Replace *ClassifyImage.cs* with the following code

```

using System;
using System.IO;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text.Json;
using System.Threading.Tasks;
using Azure.Storage.Blobs;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Logging;

namespace FN_Isolated_Vision_Analysis
{
    public class ClassifyImage
    {
        private readonly HttpClient _httpClient;

        public ClassifyImage(HttpClient httpClient)
        {
            _httpClient = httpClient;
        }

        [Function("ClassifyImage")]
        public async Task RunAsync(
            [BlobTrigger("uploaded/{name}", Connection = "BLOB_CONN")] Stream myBlob,
            string name,
            FunctionContext context)
        {
            var logger = context.GetLogger("ClassifyImage");
            try
            {
                logger.LogInformation($"C# Blob trigger function Processed blob\n Name:{name} \n Size: {myBlob.Length} Bytes");

                // Copy the blob to a MemoryStream so it can be reused
            }
            catch { }
        }
    }
}

```

```
logger.LogInformation($"Copying blob '{name}' to MemoryStream for further processing...");

var ms = new MemoryStream();
await myBlob.CopyToAsync(ms);
ms.Position = 0;

var result = await AnalyseImageAsync(ms, logger);

logger.LogInformation($"Is Adult: {result.adult.isAdultContent}");
logger.LogInformation($"Adult Score: {result.adult.adultScore}");
logger.LogInformation($"Is Racy: {result.adult.isRacyContent}");
logger.LogInformation($"Racy Score: {result.adult.racyScore}");

// Reset stream position for re-upload
ms.Position = 0;
logger.LogInformation($"Resetting MemoryStream position to 0 for upload...");

var destContainer = (result.adult.isAdultContent || result.adult.isRacyContent) ? "rejected" :
"accepted";
logger.LogInformation($"Destination container: {destContainer}");

await StoreBlobWithMetadata(ms, destContainer, name, result, logger);

// Note: The function instance is short-lived, and the MemoryStream will be garbage collected
after the function completes

}
catch (Exception ex)
{
    logger.LogError(ex, $"Error processing blob '{name}': {ex.Message}");
    throw;
}
}

private async Task<ImageAnalysisInfo> AnalyseImageAsync(Stream blob, ILogger logger)
{
    logger.LogInformation("AnalyseImageAsync(): Analyzing image for adult content...");

    var key = Environment.GetEnvironmentVariable("SubscriptionKey");
    var endpoint = Environment.GetEnvironmentVariable("VisionEndpoint");

    var visionServiceUrl = $"{endpoint}vision/v2.0/analyze?visualFeatures=Adult";
    logger.LogInformation($"Vision Service URL: {visionServiceUrl}");

    var request = new HttpRequestMessage(HttpMethod.Post, visionServiceUrl);
    request.Headers.Add("Ocp-Apim-Subscription-Key", key);

    request.Content = new NonDisposingStreamContent(blob);

    request.Content.Headers.ContentType = new MediaTypeHeaderValue("application/octet-stream");

    var response = await _httpClient.SendAsync(request);
    response.EnsureSuccessStatusCode();
}
```

```
var json = await response.Content.ReadAsStringAsync();
logger.LogInformation($"Response from Vision API: {json}");

return JsonSerializer.Deserialize<ImageAnalysisInfo>(json);
}

private async Task StoreBlobWithMetadata(Stream image, string containerName, string blobName,
ImageAnalysisInfo info, ILogger logger)
{
    logger.LogInformation($"StoreBlobWithMetadata(): Writing blob and metadata to
{containerName} container...");

    var connection = Environment.GetEnvironmentVariable("BLOB_CONN");
    logger.LogInformation($"Blob connection string: {connection}");

    var blobServiceClient = new BlobServiceClient(connection);
    var containerClient = blobServiceClient.GetBlobContainerClient(containerName);
    await containerClient.CreateIfNotExistsAsync();

    var blobClient = containerClient.GetBlobClient(blobName);

    // Reset stream position before upload
    image.Position = 0;
    await blobClient.UploadAsync(image, overwrite: true);

    logger.LogInformation($"Blob {blobName} uploaded to {containerName} container.");

    var metadata = new System.Collections.Generic.Dictionary<string, string>
    {
        ["isAdultContent"] = info.adult.isAdultContent.ToString(),
        ["adultScore"] = info.adult.adultScore.ToString("P0").Replace(" ", ""),
        ["isRacyContent"] = info.adult.isRacyContent.ToString(),
        ["racyScore"] = info.adult.racyScore.ToString("P0").Replace(" ", "")
    };

    logger.LogInformation($"Setting metadata: {string.Join(" ", metadata)}");

    await blobClient.SetMetadataAsync(metadata);
}

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; }
}
```

```

/// <summary>
/// Custom StreamContent that does not dispose the underlying stream,
/// to prevent your MemoryStream from being closed and disposing it
/// </summary>
public class NonDisposingStreamContent : StreamContent
{
    public NonDisposingStreamContent(Stream stream) : base(stream) { }
    protected override void Dispose(bool disposing)
    {
        // Prevent disposing the underlying stream
        base.Dispose(false);
    }
}
}

```

2. Look at the constructor for *ClassifyImage.cs*

```

public class ClassifyImage
{
    private readonly HttpClient _httpClient;

    0 references
    public ClassifyImage(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }
}

```

34. [Function([IClassifyImage])]
3. The constructor is expecting *HttpClient object* to be passed. In .NET 8, constructor parameters are injected during the service initiation in *Program.cs*
4. Open **Program.cs** and update the code like this:

```

using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

var host = new HostBuilder()
    .ConfigureFunctionsWorkerDefaults()
    .ConfigureServices(services =>
    {
        services.AddHttpClient();
    })
    .Build();

host.Run();

```

```

using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

var host = new HostBuilder()
    .ConfigureFunctionsWorkerDefaults()
    .ConfigureServices(services =>
    {
        services.AddHttpClient();
    })
    .Build();

host.Run();

```

5. Save changes

Module 2 – Create Azure AI Vision Service

1. In a browser, go to Azure portal
2. Open Cloud Shell
3. Set variables (replace them with your resource values) for Azure CLI commands

```
RESOURCE_GROUP="[your-resource-group]"
SERVICE_NAME="[your-short-name]-cvs-vision-analysis"
LOCATION="australiaeast"
```

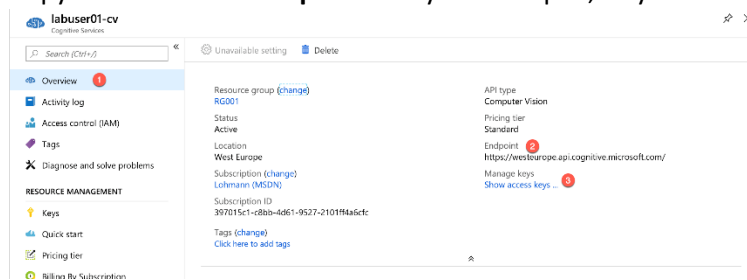
4. Create computer Vision Service

```
az cognitiveservices account create \
--name $SERVICE_NAME \
--resource-group $RESOURCE_GROUP \
--kind ComputerVision \
--sku S1 \
--location $LOCATION \
--yes
```

5. To retrieve API Keys, you can run

```
az cognitiveservices account keys list \
--name $SERVICE_NAME \
--resource-group $RESOURCE_GROUP
```

6. Go to Azure Portal and select your newly created Computer Vision Service
7. Copy the *URL* under **Endpoint** into your Notepad, so you can easily retrieve it in a moment



- 35.
8. If you don't have the API key from the step above using Cloud Shell, click **Show access keys**. Copy the KEY1 and paste it into your Notepad

Module 4 – Add Application Settings in VS

1. Go back to VS
2. Open local.settings.json file
3. Add 2x new application settings for Computer Vision Service after "FUNCTIONS_WORKER_RUNTIME" entry


```
"SubscriptionKey": "<your-subscription-key>",  
"VisionEndpoint": "<your-endpoint-url>"
```

4. Replace the values with the ones you've copied to Notepad

```
{  
  "IsEncrypted": false,  
  "Values": {  
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",  
    "BLOB_CONN": "UseDevelopmentStorage=true",  
    "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated",  
    "SubscriptionKey": "7c43569d67ef5ac3b0aa23",  
    "VisionEndpoint": "https://australiaeast.api.cognitive" 36.  
  }  
}
```

5. Save changes

Module 5 – Test Your Function Locally

1. Download sample image files from [here](#) and extract to your working folder
2. Run your Function locally like you did previously and upload a sample file
- 3.

Summary



ACHIEVEMENTS

After you have completed the exercise, you are now able to:

- ✓ Create App Service and underlying infrastructure App Service Plan
- ✓ Access and navigate the Kudu (Advanced Tools) interface for Azure App Service
- ✓ Locate and inspect environment variables relevant to your deployed .NET application
- ✓ Use the Kudu Debug Console to explore the file system, including the site\wwwroot directory where your app is hosted
- ✓ Understand how your .NET app is deployed and served via IIS on Windows-based App Service
- ✓ View and interpret deployment logs to troubleshoot and verify deployment status
- ✓ Execute basic PowerShell or CMD commands within the App Service environment for diagnostics and inspection

Exercise 4- Deploy Your Function (15min)

Topics	In this exercise, we will cover the following topics. <ul style="list-style-type: none"> Publish Vision Analysis functionality using Azure Function
Duration	<ul style="list-style-type: none"> 15 min
Tool(s)	<ul style="list-style-type: none"> Azure portal Visual Studio 2022
Subscription	[selected subscription]
Resource Group	[selected RG]

Module 1 – Publish Your Function

1. Publish your Function like you did previously

Module 2 – Add App Settings for Computer Vision Service

1. In *Azure portal*, go to your deployed **Function App**
2. In the left menu, select **Environment variables** under *Settings*
3. Delete the extra **AzureWebJobsStorage_*** entries
4. Under **Application settings**, add or update the following keys:
 - a. **SubscriptionKey** - set to your Azure Computer Vision Service key
 - b. **VisionEndpoint** – set to your Azure Computer Vision Service endpoint
5. Click **Apply** and save changes

Module 3 – Test Your Function

1. Open **Log Stream** in your Function App
2. Open a new browser and go to your Storage Account
3. Select “uploaded” container and upload a sample file
4. You will see a successful execution
5. Try uploading various files

Summary



ACHIEVEMENTS

After you have completed the exercise, you are now able to:

- ✓ Publish your Function App in Azure
- ✓ Test your Function App using Log Stream

Exercise 5 (Optional)- Manage Secrets (20min)

Scenario	<p>During the lab exercise, we have hard-coded the secrets and sensitive information in the config file and Application Configurations in App Service/Function App. That's not secure and it's easy for anyone to access your secrets.</p> <p>You need to implement a solution to secure and manage your secrets and sensitive information.</p> <p>In Azure, Azure Key Vault is a service that provides centralized secrets management, with full control over access policies and audit history.</p> <p>You will create Azure Key Vault to store secrets and sensitive information used for your application, and reference the values in Key Vault from your application</p>
Topics	<p>In this exercise, we will cover the following topics.</p> <ul style="list-style-type: none"> • Create Key Vault for Secrets • Create Secrets • Update Application Configuration Settings for Key Vault Reference • Test Your Function
Duration	<ul style="list-style-type: none"> • 20 min
Tool(s)	<ul style="list-style-type: none"> • Azure portal • Visual Studio 2022
Reference(s)	<ul style="list-style-type: none"> • Use Key Vault References as App Settings - Azure App Service Microsoft Learn • Managed Identities - Azure App Service Microsoft Learn
Subscription	[selected subscription]
Resource Group	[selected RG]

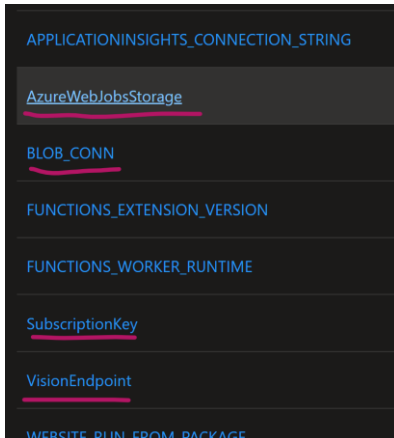
Module 1 – Create Key Vault for Secrets

1. Create a new Key Vault by following the instructions on [Use Key Vault References as App Settings - Azure App Service | Microsoft Learn](#)
2. In the “Grant your app access to a key vault” above, complete the step **1**, **2**, and **3** (managed identity)
3. Read the “Understand source app settings from Key Vault” section in the above instruction page and understand **Key Vault reference feature**
4. What it does is instead of storing your secrets in your configuration file or Application configurations in App Service/Function App, you are only to refer the value at runtime. You don't need to program anything to connect to your Key Vault resource in code. Key Vault Reference feature does it automatically for you

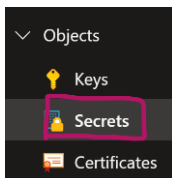
Module 2 – Create Secrets in Key Vault

1. Go to your Function App in Azure portal
2. Select **Environment variables** under the *Settings* section

- Remember we have added a number of configuration settings for Function App. All of them contain secrets or sensitive information (i.e. server URL)



- You need to protect those values securely and manage them from Key Vault
- Go to your *Key Vault* in another browser tab. Make sure you are assigned with the correct RBAC permission to create keys in Key Vault (Q: What is the RBAC role you need to be added to?)
- Select **Secrets** under the *Objects* section



- Click + Generate/Import button
- Enter the following details

Item	Description
Upload option	Manual
Name	dev-workshop-fnsta-connstr
Secret value	(copy the value for “ <i>AzureWebJobsStorage</i> ” setting in your Function Application Configuration)
Content type	
Set activation date	
Set expiration date	
Enabled	Yes
Tags	



Naming Convention for Secrets

A name of your secret must be:

- Easy to identify what it is and descriptive enough
- Easy to read

Try following the suggested naming conventions, if you don't already have your organisation's standard

<Environment>-<AppName>-<ResourceType>-<Purpose>

- Environment*: dev, test, qa, prod
- AppName*: Short name of the app or service
- ResourceType*: kv (Key Vault), sa (Storage Account), sql, cosmos, etc.

- **Purpose:** What the secret is for, e.g., connstr, apikey, clientsecret

e.g.

dev-ordersapp-sa-connstr for Storage connection string for Orders App (Dev), qa-webapp-kv-apiKey for API key stored in Key Vault for QA Web App

- Click **Create** button
- Repeat the step 8 & 9 for **BLOB_CONN**, **SubscriptionKey**, **VisionEndpoint** settings
 - BLOB_CONN – the value for Blob connection (at the moment, it's the same as "AzureWebJobsStorage")
 - SubscriptionKey – the value of Azure Computer Vision service key
 - VisionEndpoint – the value of Azure Computer Vision service endpoint
- You have created 4x secrets in Key Vault

e.g.

Name	Type	Status
dev-azdevelopment-cvision-svc-endpoint		✓ Enabled
dev-azdevelopment-cvision-subkey		✓ Enabled
dev-azdevelopment-blob-connstr		✓ Enabled
dev-azdevelopment-fnsta-connstrg		✓ Enabled

Module 3 – Update Application Configuration Settings for KV Reference

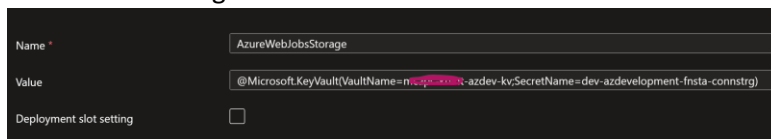


Key Vault Reference in Application Configuration

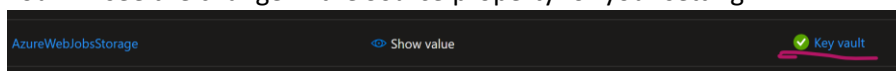
A reference format to connect to Key Vault secrets is like this:

[@Microsoft.KeyVault\(VaultName=myvault;SecretName=mysecret\)](#)

- Go to your *Function App* and select **Environment variables** under the Settings section
- Select **AzureWebJobsStorage** setting
- Replace the existing value with KV reference using the above format
- It looks something like this:



- Click **Apply** and **Apply** button again to update the App settings
- You will see the change in the *Source* property for your setting



- The green icon indicates KV reference is successfully created for you
- Repeat the steps for the rest of 3 secrets

AzureWebJobsStorage	Show value	✓ Key vault
BLOB_CONN	Show value	✓ Key vault
FUNCTIONS_EXTENSION_VERSION	Show value	App Service
FUNCTIONS_WORKER_RUNTIME	Show value	App Service
SubscriptionKey	Show value	✓ Key vault
VisionEndpoint	Show value	✓ Key vault

Module 4 – Test your Function

1. Test and validate if your Function still works as before
2. Upload a sample image(s) to Blob Storage
3. You can run Log Stream for your Function to see the live logs as well
4. Everything works just like before? Great!
5. This is the end of lab exercises

Summary



ACHIEVEMENTS

After you have completed the exercise, you are now able to:

- ✓ Secure secret management
 - No hardcoded secrets in code or config files
 - Secrets like connection strings, API keys, and passwords are stored securely in Azure Key Vault
 - Access is controlled via Azure RBAC
- ✓ Separate secret management away from the application development – secrets have its own lifecycle away from the application development
- ✓ If a secret in Key Vault is updated, the app automatically picks up the new value without redeployment
- ✓ This reduces downtime and manual updates