**About Azure Functions**

**Web-Queue-Worker Arhitecture**

Diagram

Description automatically generated

* Azure Functions is a **serverless compute service** that enables you to run code on-demand without having to explicitly provision or manage infrastructure.
* Use Azure Functions to run a script or piece of code in response to a **variety of events**.
* Functions can make development even more productive, and you can use your development language of choice, such as C#, Java, Node.js, Python or PHP.

The following service integrations are supported by Azure Functions:

|  |  |
| --- | --- |
| Azure Cosmos DBAzure Event HubsAzure Event GridAzure Notification HubsAzure Service Bus (queues and topics)Azure Storage (blob, queues, and tables)On-premises (using Service Bus) |  |

The following are a common, but by no means exhaustive, set of scenarios for Azure Functions.

|  |  |
| --- | --- |
| **If you want to...** | **then...** |
| **Build a web API** | Implement an endpoint for your web applications using the **HTTP trigger** |
| **Process file uploads** | Run code when a file is uploaded or changed in **blob storage.** |
| **Build a serverless workflow** | Chain a series of functions together using **durable functions.** |
| **Respond to database changes** | Run custom logic when a document is created or updated in Cosmos DB |
| **Run scheduled tasks** | Execute code on pre-defined timed intervals (Timer trigger) |
| **Create reliable message queue systems** | Process message queues using Queue Storage, Service Bus, or Event Hubs |
| **Analyze IoT data streams** | Collect and process data from IoT devices |
| **Process data in real time** | Use Functions and SignalR to respond to data in the moment |

## **How much does Functions cost?**

Azure Functions has following kinds of pricing plans:

* **Consumption plan**
  + Billing is based on number of executions, execution time, and memory used. Billing is aggregated across all functions within a function app.
  + When your function runs, Azure provides all of the necessary computational resources.
  + You don't have to worry about resource management, and you only pay for the time that your code runs.
  + Your app scales additional instances of the Functions host when needed to handle load, and scaled down when code stops running.
  + Every execution can use max up to **1.5GB Memory** and **One CPU.**
* **App Service plan**
  + Run your functions just like your web, mobile, and API apps.
  + When you are already using App Service for your other applications, you can run your functions on the same plan at no additional cost.
* **Elastic Premium plan**
  + Premium plan provides features, such as premium compute instances(one core, two core, and four core instances), Predictable Pricing, the ability to keep instances warm indefinitely, Unlimited execution duration, and VNet connectivity.
  + In the premium plan, your plan size will determine the available memory and CPU for all apps in that plan on that instance.

**Creating an Azure Function in Portal**

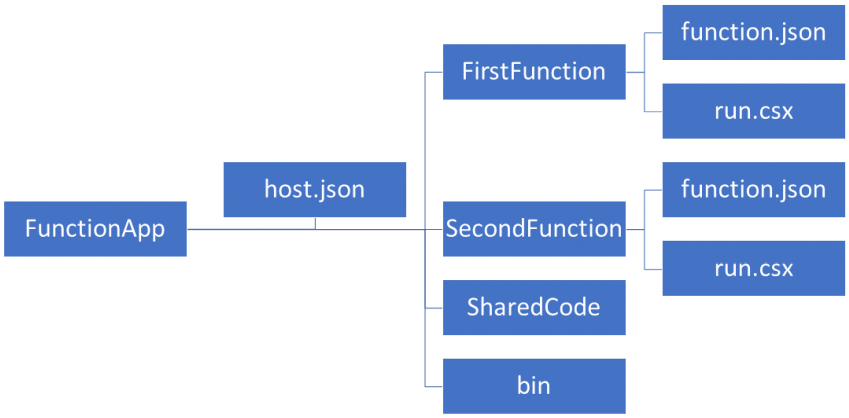
**Create a Function App**

1. +Create a resource 🡪 Compute 🡪 Function App
2. App name = DssDemoFunctions, Hosting Plan: Consumption Plan, Runtime Stack = .NET Core, Storage Account: <Create New> 🡪 Create

**Create a Function**

1. Function App 🡪 Click Functions + 🡪 **HTTP trigger**,
2. Name = SayHello,
3. Authorization level = Function / Anonymous / Admin

Authorization level controls whether the function requires an **API key** and which key to use; Function uses a function key; Admin uses your master key. The function and master keys are found in the **'keys'** management panel on the portal, when your function is selected.



**Note:** The host.json file contains runtime-specific configurations and is in the root folder of the function app. A bin folder contains packages and other library files that the function app requires.

**Following is the function auto generated in C#**

#r "Newtonsoft.Json"

using System.Net;

using Microsoft.AspNetCore.Mvc;

using Microsoft.Extensions.Primitives;

using Newtonsoft.Json;

public static async Task<IActionResult> **Run**(**HttpRequest** req, ILogger log)

{

log.LogInformation("C# HTTP trigger function processed a request.");

string name = req.Query["name"];

string requestBody = await new StreamReader(req.Body).ReadToEndAsync();

dynamic data = JsonConvert.DeserializeObject(requestBody);

name = name ?? data?.name;

return name != null

? (ActionResult)new OkObjectResult($"Hello, {name}")

: new BadRequestObjectResult("Please pass a name on the query string or in the request body");

}

**Test the function:**

1. In your new function, click **</> Get function URL** and copy the **Function URL.**

https://dssdemofunc.azurewebsites.net/api/HttpTriggerJS1?**code**=HaAojLFnYG05Fa0hMtocj7ymoCExasRWW64BaWtbQGx/O9DvSoDv8A==

1. Paste the URL for the HTTP request into your browser's address bar. Append the query string &name=<yourname> to this URL and execute the request.
2. View the function logs at the bottom of the screen (click up arrow at the bottom of the screen).
3. You can make changes (specially to context.res.body) to the function as needed and test the same.
4. You can also test the function from Right Handside panel: Test (expand to test)

**Timer Triggered Function**

1. Function App 🡪 Click Functions **+** 🡪 Choose a template: **TimerTrigger – C#** 🡪 Name your function = TimerTriggerCSharp, Schedule= 0 \*/1 \* \* \* \* (CRON expression that schedules your function to run every minute) 🡪 Create
2. If required we can update the **timer schedule**: Integrate tab 🡪

{seconds} {minutes} {hour} {day} {month} {dayofweek}

* To trigger once every hour = 0 0 \*/1 \* \* \*
* To trigger once every five minutes: 0 \*/5 \* \* \* \*
* To trigger once at the top of every hour: "0 0 \* \* \* \*
* To trigger once every two hours: 0 0 \*/2 \* \* \*
* To trigger once every hour from 9 AM to 5 PM: 0 0 9-17 \* \* \*
* To trigger At 9:30 AM every day: 0 30 9 \* \* \*
* To trigger At 9:30 AM every weekday: 0 30 9 \* \* 1-5

1. When a timer trigger function is invoked, the [timer object](https://github.com/Azure/azure-webjobs-sdk-extensions/blob/master/src/WebJobs.Extensions/Extensions/Timers/TimerInfo.cs) is passed into the function. The following JSON is an example representation of the timer object.

{

"Schedule":{ },

"ScheduleStatus": {

"Last":"2016-10-04T10:15:00.012699+00:00",

"Next":"2016-10-04T10:20:00+00:00"

},

"IsPastDue":false

}

**Example:**

using System;

public static void Run(**TimerInfo** myTimer, TraceWriter log)

{

if (myTimer.IsPastDue)

{

log.Info("Timer is running late!");

}

log.Info(myTimer.ScheduleStatus.Last.ToString());

log.Info($"C# Timer trigger function executed at: {DateTime.Now}");

}

**Queue Trigger - Input and Output Parameter Binding**

Go to Storage Account and Create a container, Name=employees and upload files 1.txt and 2.txt (1 and 2 are employee ids when you are posting a json object into queue)

**Run.csx**

using System;

public class Employee

{

public int Id {get;set;}

public string Name {get;set;}

}

public static void Run(Employee emp, string empDetails, **out** string employeeUpdatedDetails, ILogger log)

{

log.LogInformation($"C# Queue trigger function processed: {emp.Id} - {emp.Name}");

log.LogInformation(empDetails);

employeeUpdatedDetails = empDetails + " (Name=" + emp.Name + ")";

}

**Function.json**

{

"bindings": [

{

"name": "emp",

"type": "queueTrigger",

"direction": "in",

"queueName": "myqueue-items",

"connection": "AzureWebJobsStorage"

},

{

"name": "empDetails",

"direction": "in",

"type": "blob",

"path": "employees/{Id}.txt",

"connection": "AzureWebJobsStorage"

},

{

"name": "employeeUpdatedDetails",

"direction": "**out**",

"type": "blob",

"path": "employees/{Name}.txt",

"connection": "AzureWebJobsStorage"

}

]

}

**Connect to SQL Database**

1. Create an SQL Database and copy its connection string.
2. Navigate to your function app you created
3. Select **Platform features** 🡪 **Application settings.**
4. Scroll down to **Connection strings** and add a connection string "sqldb\_connection"
5. Save
6. In your function app, select the timer-triggered function

**#r "System.Configuration"**

**#r "System.Data"**

using System;

using System.Configuration;

using System.Data.SqlClient;

public class **Employee**

{

public int Id{get;set;}

public string Name {get;set;}

}

public static void Run(**Employee** emp, ILogger log, ExecutionContext context)

{

var str = **Environment.GetEnvironmentVariable**("sqldb\_connection");

**// Note: Functions written in Portal cannot read data from ConnectionStrings Section. They can only read from ApplicationSettings Sections**

log.LogInformation(str);

using (SqlConnection conn = new SqlConnection(str))

{

conn.Open();

var text = $"insert into Emp values({emp.Id},'{emp.Name}')";

using (SqlCommand cmd = new SqlCommand(text, conn))

{

// Execute the command and log the # rows affected.

var rows = cmd.ExecuteNonQuery();

log.LogInformation($"{rows} rows were inserted");

}

}

}

**Writing Function in Visual Studio.NET**

1. In Azure Portal Create Storage Account
2. Create a New Project
   * VS.NET 🡪 File 🡪 New Project 🡪 Cloud 🡪 Azure Functions, Name="DemoFunctionApp"

Graphical user interface, text, application

Description automatically generated

1. Edit the function as below with "FUNCTIONS\_WORKER\_RUNTIME": "dotnet-isolated"

using System;

using Microsoft.Azure.Functions.Worker;

using Microsoft.Extensions.Logging;

namespace DemoFunctionApp

{

      public class Employee

      {

            public int Id { get; set; }

            public string Name { get; set; }

      }

      public static class StorageQueueDemo

      {

**[Function("StorageQueueDemo")]**

            [BlobOutput(blobPath: "employees/{Name}.txt")]

            public static string Run(

                [QueueTrigger("myqueue-items", Connection = "")] Employee emp,

                [BlobInput(blobPath: "employees/{Id}.txt")] string empDetails,

                FunctionContext executionContext)

            {

                  var logger = executionContext.GetLogger("StorageQueueDemo");

                  logger.LogInformation($"C# Queue trigger function processed: {emp.Id} - {emp.Name}");

                  logger.LogInformation(empDetails);

                  logger.LogInformation("This is v2");

                  return empDetails + " (Name=" + emp.Name + ")";

            }

      }

}

1. Run the function locally.
2. Goto Azure Portal and post the message in Queue by name "myqueue-items".
3. See that the new blob as created in employees container.
4. Publish the Azure Function

Note: Edit the function as below with "FUNCTIONS\_WORKER\_RUNTIME": "dotnet" (In-Proc)

[FunctionName("StorageQueueDemo")]

public void Run(

[QueueTrigger("emp-queue", Connection = "AzureWebJobsStorage")]Employee emp,

[Blob("resumes/{name}.txt", FileAccess.Read)] string resume,

[Blob("resumes/{id}.txt", FileAccess.Write)] out string newfile,

ILogger log)

{

log.LogInformation($"C# Queue trigger function processed: {emp.Id} - {emp.Name}");

newfile = "This is info about person with id=" + emp.Id;

log.LogInformation(resume);

}

Note:

**Input/Output Parameters in In-Process and Isolated:**

<https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob-output>

**Azure Durable Functions (Overview)**

Durable Functions are an **extension** of Azure Functions that lets you write **stateful functions** in a serverless environment.

The extension lets you define **stateful workflows** by writing **orchestrator functions** and **stateful entities** by writing **entity functions** using the Azure Functions programming model.

Behind the scenes, the extension manages state, checkpoints, and restarts for you, allowing you to focus on your business logic.

**Benefits:**

* You can define your workflows in code. Unlike in Logic Apps No JSON schemas or designers are needed.
* Other functions can be called both **synchronously and asynchronously**. Output from called functions can be saved to local variables.
* Progress is automatically checkpointed when the function awaits. Local state is never lost when the process recycles or the VM reboots.

**Durable Function scenario**

1. Function Chaining
2. Fan-out/fan-in
3. Async HTTP APIs
4. Monitoring
5. Human Interaction
6. Aggregator (stateful entities)

Function chaining refers executing a sequence of functions in a particular order. Often, the output of one function needs to be applied to the input of another function

