# 1. **Azure Functions** (My first HttpTrigger Azure Function)

Azure Functions is a **serverless compute service** that allows you to run **event-triggered** code without explicitly provisioning or managing infrastructure. Functions are small units of code that can be written in various languages, and they respond to events and execute accordingly.

Let's start with a simple example in **C#**. This is a Http Triggered function, it's to say, a function that responds to **HTTP GET** and **POST** requests:

**1. Create a new Azure Function**:
Go to the Azure portal.
Click on "Create a resource" and search for "Function App."
Fill in the required details like subscription, resource group, and function app name.

**2. Create a Function**:
Inside your Function App, click on the "+" button next to "Functions."
Select "HTTP trigger" as the template. This creates a function that can be triggered via an HTTP request.

**3. Write your Function:**

```
using System.IO;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;

public static class SimpleFunction
{
   [FunctionName("SimpleFunction")]
   public static async Task<IActionResult> Run(
      [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = null)]
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");
    }
}
```

This simple function takes a **name** as a parameter (either from the query string or request body) and returns a greeting.

**Source Code explained**

This is a simple Azure Functions code written in **C#** using the Azure Functions runtime and the **ASP.NET Core framework**. Let me break it down for you:

**Namespace and Using Statements:**

```
using System.IO;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;
```

These lines include necessary namespaces for working with input/output streams, handling HTTP requests and responses, logging, and JSON serialization.

**Class Declaration:**

**public static class** SimpleFunction

Defines a static class named SimpleFunction.

**Function Method:**

```
[FunctionName("SimpleFunction")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = null)]
HttpRequest req, ILogger log)
```

The method is marked with the [FunctionName] attribute, specifying the name of the Azure Function as "SimpleFunction".

It has an HttpTrigger attribute, meaning it's triggered by HTTP requests. The allowed methods are "**get**" and "**post**", and the authorization level is set to "**Function**".

It takes an HttpRequest object for handling HTTP requests and an ILogger for logging. Function Body:

**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");

**Logs** an information message about processing the request.

Retrieves the "**name**" **parameter** from the query string of the HTTP request.

Reads the request body and deserializes it from **JSON** using **Newtonsoft.Json**.

If the "name" parameter is not present in the query string, it tries to get it from the request body.

Responds with an **OkObjectResult** containing a greeting if a name is provided, or a BadRequestObjectResult asking for a name if none is provided.

In summary, this function is a basic **HTTP-triggered Azure Function** that expects a "name" parameter either in the **query string** or the **request body** and responds with a greeting or a bad request message accordingly.

---

## 2. Azure Functions (Triggers with a sample for each)

Azure Functions supports various triggers to initiate the execution of functions. Here are some common triggers along with a sample for each:

**1. Timer Trigger**:

Triggered based on a schedule. This function is triggered every 5 minutes**.**

[FunctionName("TimerTriggerFunction")]
public static void **Run**(
[TimerTrigger("0 */5 * * * *")] TimerInfo myTimer,

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

**2. Queue Trigger**:

Triggered when a message is added to a queue. This function listens to a queue named "myqueue-items."

```
[FunctionName("QueueTriggerFunction")]
public static void Run(
    [QueueTrigger("myqueue-items", Connection = "AzureWebJobsStorage")] string myQueueItem,
    ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed message: {myQueueItem}");
}
```

**3. Blob Trigger**:

Triggered when a new or updated blob is detected in a storage container. This function is triggered when a blob is added or modified in the "mycontainer" container.

```
[FunctionName("BlobTriggerFunction")]
public static void Run(
    [BlobTrigger("mycontainer/{name}", Connection = "AzureWebJobsStorage")] Stream myBlob,
    string name,
    ILogger log)
{
    log.LogInformation($"C# Blob trigger function processed blob\n Name:{name} \n Size: {myBlob.Length} Bytes");
}
```

**4. Service Bus Trigger**:

Triggered when a message is received from Azure Service Bus. This function listens to a Service Bus queue named "myqueue".

```
[FunctionName("ServiceBusTriggerFunction")]
public static void Run(
    [ServiceBusTrigger("myqueue", Connection = "ServiceBusConnection")] string myQueueItem,
```

```
    ILogger log)
{
    log.LogInformation($"C# Service Bus trigger function processed message:
{myQueueItem}");
}
```

**5. Cosmos DB Trigger**:

Triggered when documents within a Cosmos DB collection are created or modified. This function is triggered when documents are added or modified in the "MyCollection" collection of the Cosmos DB.

```
[FunctionName("CosmosDBTriggerFunction")]
public static void Run([CosmosDBTrigger(databaseName: "MyDatabase",
 collectionName: "MyCollection",ConnectionStringSetting = "CosmosDBConnection",
 LeaseCollectionName = "leases", CreateLeaseCollectionIfNotExists = true)]
IReadOnlyList<Document> documents, ILogger log)
{
    if (documents != null && documents.Count > 0)
    {
        log.LogInformation($"C# Cosmos DB trigger function processed
{documents.Count} documents.");
        log.LogInformation($"First document Id: {documents[0].Id}");
    }
}
```

**6. Event Hub Trigger**:

Triggered when events are sent to an Azure Event Hub. This function is triggered when events are sent to the "myeventhub" Event Hub.

```
[FunctionName("EventHubTriggerFunction")]
public static void Run([EventHubTrigger("myeventhub", Connection =
EventHubConnection")] string[] events, ILogger log)
{
    foreach (var eventData in events)
    {
        log.LogInformation($"C# Event Hub trigger function processed event:
{eventData}");
    }
}
```

**7. Event Grid Trigger**:

´Triggered when an event is published to Azure Event Grid. This function is triggered when an event is published to the associated Event Grid topic.

```
[FunctionName("EventGridTriggerFunction")]
public static void Run([EventGridTrigger] EventGridEvent eventGridEvent,
    ILogger log)
{
    log.LogInformation($"C# Event Grid trigger function processed event:
    {eventGridEvent.Data}");
}
```

**8. GitHub Webhook Trigger**:

Triggered when a GitHub webhook event is received. This function is triggered when a GitHub webhook event is received (**requires appropriate GitHub webhook setup**).

```
[FunctionName("GitHubWebhookTriggerFunction")]
public static async Task<IActionResult>
Run([HttpTrigger(AuthorizationLevel.Function, "post", Route = null)] HttpRequest
req, ILogger log)
{
    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    var data = JsonConvert.DeserializeObject<GitHubWebhookData>(requestBody);

    log.LogInformation($"C# GitHub Webhook trigger function processed event:
{data.Action}");

    return new OkResult();
}
```

**9. Twilio SMS Trigger**:

Triggered when an SMS is sent to a Twilio phone number. This function is triggered when an SMS is received by a Twilio phone number.

```
[FunctionName("TwilioSMSTriggerFunction")]
public static async Task<IActionResult> Run([TwilioSmsTrigger("SmsReceived",
"TwilioAccountSidSetting", "TwilioAuthTokenSetting")] TwilioSmsMessage sms,
    ILogger log)
{
    log.LogInformation($"C# Twilio SMS trigger function processed message:
{sms.Body}");

    // Your processing logic here

    return new OkResult();
}
```

**10. Microsoft Graph Webhook Trigger**:

Triggered when an event occurs in Microsoft 365 services (e.g., Outlook, OneDrive). This function is triggered when an event occurs in Microsoft 365, such as receiving an email.

```
[FunctionName("MicrosoftGraphWebhookTriggerFunction")]
public static async Task<IActionResult> Run([EventGridTrigger] EventGridEvent
eventGridEvent, ILogger log)
{
   if (eventGridEvent.EventType == "microsoft.graph.mailReceived")
   {
      log.LogInformation($"C# Microsoft Graph Webhook trigger function processed
mailReceived event.");
   }

   // Your processing logic here

   return new OkResult();
}
```

**11. Azure Durable Task Orchestration**:

Orchestrations enable you to write workflows that can coordinate the execution of multiple functions. This example shows an orchestrator function calling three activity functions in parallel.

```
[FunctionName("OrchestrationFunction")]
public static async Task<List<string>> RunOrchestrator(
   [OrchestrationTrigger] DurableOrchestrationContext context)
{
   var outputs = new List<string>();

   // Replace "hello" with the name of your Durable Activity Function.
   outputs.Add(await context.CallActivityAsync<string>
("OrchestrationFunction_Hello", "Tokyo"));
   outputs.Add(await context.CallActivityAsync<string>
("OrchestrationFunction_Hello", "Seattle"));
   outputs.Add(await context.CallActivityAsync<string>
("OrchestrationFunction_Hello", "London"));

   // returns ["Hello Tokyo!", "Hello Seattle!", "Hello London!"]
   return outputs;
}

[FunctionName("OrchestrationFunction_Hello")]
```

```
public static string SayHello([ActivityTrigger] string name, ILogger log)
{
   log.LogInformation($"Saying hello to {name}.");
   return $"Hello {name}!";
}
```

**12. IoT Hub Trigger**:

Triggered when a message is sent to Azure IoT Hub. This function is triggered when a message is sent to the "messages/events" endpoint of Azure IoT Hub.

```
[FunctionName("IoTHubTriggerFunction")]
public static void Run([IoTHubTrigger("messages/events", Connection =
"IoTHubConnection")]EventData message, ILogger log)
{
   var messageBody = Encoding.UTF8.GetString(message.Body.Array);
   log.LogInformation($"C# IoT Hub trigger function processed message:
{messageBody}");
}
```

**13. SignalR Trigger**:

Triggered when a message is sent to a SignalR Service hub. This function is triggered when a message is sent to the "messages" hub in SignalR Service.

```
[FunctionName("SignalRTriggerFunction")]
public static void Run([SignalRTrigger("messages", "connections",
"mySignalRConnection")] SignalRMessage message, ILogger log)
{
   var payload = JsonConvert.DeserializeObject<MyPayloadType>
(message.Payload.ToString());
   log.LogInformation($"C# SignalR trigger function processed message:
{payload.Message}");
}
```

**14. Azure Key Vault Trigger:**

Triggered when a secret is added or updated in Azure Key Vault. This function is triggered when the specified secret is added or updated in the Azure Key Vault.

```
[FunctionName("KeyVaultTriggerFunction")]
public static void Run([KeyVaultSecretTrigger("MyKeyVaultName",
"MySecretName")] string secretValue, ILogger log)
{
   log.LogInformation($"C# Key Vault trigger function processed secret:
{secretValue}");
```

```
}
```

**15. Azure SignalR Service Trigger:**

Triggered when a message is sent to an Azure SignalR Service hub.

```
[FunctionName("SignalRServiceTriggerFunction")]
public static void Run([SignalRTrigger("messages", "connections",
"mySignalRConnection")] SignalRMessage message, ILogger log)
{
   var payload = JsonConvert.DeserializeObject<MyPayloadType>
(message.Payload.ToString());
   log.LogInformation($"C# SignalR Service trigger function processed message:
{payload.Message}");
}
```

---

## 3. Azure Functions (advance concepts)

Functions is a serverless computing service provided by Microsoft Azure, allowing you to run event-triggered code without explicitly provisioning or managing infrastructure. Here are some advanced concepts:

**Bindings:**

Azure Functions use bindings to simplify **input** and **output** integration with other Azure services. There are various bindings like Blob Storage, Azure Cosmos DB, Event Hubs, etc. You can define these bindings in your function's code and configuration.

**Sample 1 using Azure Blob Storage binding** in C#:

```
public static void Run([BlobTrigger("mycontainer/{name}")] Stream myBlob, string
name, ILogger log)
{
   log.LogInformation($"Blob trigger function processed blob\n Name:{name} \n
Size: {myBlob.Length} Bytes");
}
```

**Sample 2 using Azure Queue Storage binding** in C#. In this example:

The QueueTriggerFunction function is triggered by an HTTP request, and it adds a **message** to the Azure Queue Storage named "myqueue" using the IAsyncCollector<string> binding.

The QueueProcessFunction function is triggered by messages added to the "myqueue" queue. It logs the message and performs any processing logic you desire.

This demonstrates the simplicity of using Azure Queue Storage binding to connect your functions with Azure Storage queues, making it easy to integrate and process messages.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;

public static class QueueTriggerFunction
{
   [FunctionName("QueueTriggerFunction")]
   public static async Task<IActionResult> Run(
      [HttpTrigger(AuthorizationLevel.Function, "post", Route = null)] HttpRequest
req, [Queue("myqueue")] IAsyncCollector<string> queueCollector, ILogger log)
   {
      log.LogInformation("C# HTTP trigger function processed a request.");

      string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
      dynamic data = JsonConvert.DeserializeObject(requestBody);

      string message = data?.message;
      await queueCollector.AddAsync(message);

      return new OkObjectResult($"Message '{message}' added to the queue
successfully.");
   }

   [FunctionName("QueueProcessFunction")]
   public static void ProcessQueueMessage(
      [QueueTrigger("myqueue")] string message,
      ILogger log)
   {
      log.LogInformation($"Processing queue message: {message}");
      // Add your processing logic here
   }
}
```

**Durable Functions:**

This is an extension of Azure Functions that allows you to write **stateful** functions in a **serverless** environment. Durable Functions enable workflows where functions can have input and output over time, handle timeouts and errors, and can be composed

together in complex patterns.

**Sample 1 of a durable orchestrator function** in C#:

Tha "Orchestrator_Function" calls the "Hello" functions as activities to execute in parallel.

```
[FunctionName("Orchestrator_Function")]
public static async Task RunOrchestrator([OrchestrationTrigger]
DurableOrchestrationContext context)
{
   var outputs = new List<string>();

   outputs.Add(await context.CallActivityAsync<string>("Hello", "Tokyo"));
   outputs.Add(await context.CallActivityAsync<string>("Hello", "Seattle"));
   outputs.Add(await context.CallActivityAsync<string>("Hello", "London"));

   return outputs;
}

[FunctionName("Hello")]
public static string SayHello([ActivityTrigger] string city, ILogger log)
{
   log.LogInformation($"Saying hello to {city}.");
   return $"Hello {city}!";
}
```

**Sample 2 of a durable orchestrator function** in C#:

HttpStart is an HTTP-triggered function that initiates the orchestration.

OrchestratorFunction is the main function that defines the workflow using tasks executed in sequence and in parallel.

ActivityFunction represents the individual tasks that can be executed by the orchestrator.

Make sure to adjust the function names and logic according to your specific use case. You can use the HttpStart function's endpoint to trigger the orchestration via an HTTP request.

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
```

```
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;
using Microsoft.Azure.WebJobs.Extensions.DurableTask;

public static class DurableFunctionExample
{

   [FunctionName("HttpStart")]
   public static async Task<HttpResponseMessage> HttpStart(
      [HttpTrigger(AuthorizationLevel.Function, "get", "post")] HttpRequestMessage
req, [DurableClient] IDurableOrchestrationClient starter, ILogger log)
   {
      // Function input comes from the request content.
      string instanceId = await starter.StartNewAsync("OrchestratorFunction", null);

      log.LogInformation($"Started orchestration with ID = '{instanceId}'.");

      return starter.CreateCheckStatusResponse(req, instanceId);
   }

   [FunctionName("OrchestratorFunction")]
   public static async Task RunOrchestrator([OrchestrationTrigger]
IDurableOrchestrationContext context)
   {
      // Define tasks (task3 and task4)
      var task1 = await context.CallActivityAsync<string>("ActivityFunction", "Task 1");
      var task2 = await context.CallActivityAsync<string>("ActivityFunction", "Task 2");

      // Parallel execution of task3 and task4
      var parallelTasks = new List<Task<string>>
      {
         context.CallActivityAsync<string>("ActivityFunction", "Task 3"),
         context.CallActivityAsync<string>("ActivityFunction", "Task 4")
      };

      await Task.WhenAll(parallelTasks);

      var task3 = parallelTasks[0].Result;
      var task4 = parallelTasks[1].Result;

      // Compose results
      var result = $"Orchestration completed. {task1} {task2} {task3} {task4}";
      return result;
   }

   [FunctionName("ActivityFunction")]
```

```
    public static string RunActivity([ActivityTrigger] string taskName, ILogger log)
    {
        log.LogInformation($"Executing task: {taskName}");
        return $"Completed {taskName}";
    }
}
```

**Dependency Injection:**

Azure Functions support dependency injection, allowing you to inject services or dependencies into your functions. This promotes modular and testable code.

**Sample 1 using dependency injection** in C#:

```
public class MyFunction
{
    private readonly ILogger<MyFunction> _logger;

    public MyFunction(ILogger<MyFunction> logger)
    {
        _logger = logger;
    }

    [FunctionName("MyFunction")]
    public void Run([TimerTrigger("0 */5 * * * *")] TimerInfo myTimer)
    {
        _logger.LogInformation($"C# Timer trigger function executed at:
{DateTime.Now}");
    }
}
```

**Proxies:**

Azure Function Proxies allow you to define additional routes, rewrite URLs, or even combine multiple functions into a single API endpoint. It's handy for creating a façade over your functions.

Here's a simple example of Azure Function Proxies in **proxies.json**:

```
{
  "$schema": "http://json.schemastore.org/proxies",
  "proxies": {
    "proxy1": {
      "matchCondition": {
        "route": "/api/proxy1/{*restOfPath}"
      },
```

```
    "backendUri": "https://yourfunctionapp.azurewebsites.net/{restOfPath}",
    "methods": ["GET", "POST"]
   },
   "proxy2": {
    "matchCondition": {
     "route": "/api/proxy2/{*restOfPath}"
    },
    "backendUri": "https://anotherfunctionapp.azurewebsites.net/{restOfPath}",
    "methods": ["GET", "POST"]
   }
  }
}
```

In this example, we have two proxies (**proxy1** and **proxy2**). Each proxy defines a matchCondition specifying the route and a backendUri pointing to the backend function app. The methods array indicates which HTTP methods are allowed.

Remember to replace "https://yourfunctionapp.azurewebsites.net" and "https://anotherfunctionapp.azurewebsites.net" with the actual URLs of your Azure Function apps.

**Managed Identities:**

Azure Functions can use managed identities to interact with other Azure services securely. This eliminates the need for storing credentials in your code.
Auto-scaling:

Azure Functions automatically scale based on demand. Understanding how this works and configuring it effectively is crucial for handling varying workloads efficiently.

Here's a simple example of an Azure Function using **managed identities** to interact with **Azure Blob Storage**:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Blob;

public static async Task<IActionResult> Run(HttpRequest req, ILogger log)
{
   log.LogInformation("C# HTTP trigger function processed a request.");

   // Using Managed Identity to get access token
   var storageAccount =
CloudStorageAccount.FromConfigurationSetting("yourConnectionString");
```

```
    var blobClient = storageAccount.CreateCloudBlobClient();
    var container = blobClient.GetContainerReference("yourContainerName");

    // Your code to interact with Azure Blob Storage using managed identity
    var blob = container.GetBlockBlobReference("yourBlobName");
    string blobContent = await blob.DownloadTextAsync();

    return new OkObjectResult($"Blob content: {blobContent}");
}
```

In this example, replace "yourConnectionString", "yourContainerName", and "yourBlobName" with your actual Azure Storage connection string, container name, and blob name, respectively.

This code retrieves the content of a blob from Azure Blob Storage using managed identities for authentication.

**Auto-Scaling:**

For auto-scaling, Azure Functions automatically scale based on demand. You can configure this in the Azure portal or using Azure CLI. Here's an example of using Azure CLI to set the minimum and maximum instance count:

**az functionapp plan update --name <your-plan-name> --resource-group <your-resource-group> --min-instances 1 --max-instances 10**

This sets the minimum number of instances to 1 and the maximum to 10, allowing your function app to scale based on demand.

Remember to replace placeholders like <your-plan-name> and <your-resource-group> with your actual values.

**Custom Handlers:**

You can use custom handlers to run your functions in languages that are not natively supported by Azure Functions. This allows you to bring your own runtime.

you can create a simple custom handler for an Azure Functions "Hello, World!" function. Here's an example:

Create a file named **Function.cs** with the following content:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;

public static class Function
```

```
{
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = null)]
HttpRequest req)
    {
        return new OkObjectResult("Hello, World!");
    }
}
```

Now, create a **function.json** file that describes the function:

```
{
  "scriptFile": "Function.dll",
  "entryPoint": "Function.Run",
  "bindings": [
    {
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "authLevel": "function",
      "methods": ["get", "post"],
      "route": null
    },
    {
      "type": "http",
      "direction": "out",
      "name": "$return"
    }
  ]
}
```

Create a **host.json** file to specify the custom handler:

```
{
  "version": "2.0",
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[3.*, 4.0.0)"
  },
  "customHandler": {
    "description": {
      "defaultExecutablePath": "dotnet",
      "workingDirectory": "",
      "arguments": ["Function.dll"]
    }
  }
```

}

Build the project:

```
dotnet build
```

Run your function:

```
func host start
```

This will start the Azure Functions runtime with your custom handler.

Test your function by navigating to http://localhost:7071/api/<your-function-name> in your browser or using a tool like curl or Postman.

This example assumes you have the .NET SDK installed. Adjust the **Function.dll** and **Function.Run** values in function.json if your function has a different project structure or entry point.

**Triggers and Bindings Extensibility:**

You can create custom triggers and bindings to extend the capabilities of Azure Functions. This is useful when you need to integrate with services not supported out of the box.

**Security Best Practices:**

Implementing secure coding practices, managing secrets, and configuring authentication and authorization are crucial aspects. Azure Functions integrate with Azure AD for authentication and authorization.

**Monitoring and Logging:**

Azure Functions provide integration with Azure Application Insights for monitoring and logging. Understanding how to utilize these tools can help you troubleshoot and optimize your functions.
These advanced concepts can take your Azure Functions development to the next level.