* To work with Azure function first we need to install Azure function core tool
* Npm install -g azure-functions-core-tools
* Func -v gives the version of the Azure function
* Install Azure function tool extension in the VS code
* Create a folder, change directory to that folder and then run
* **Func init** will create an azure function project in the name of the folder
* Code . to open the VS code for this Azure function project
* To add a new function to the function project just type

**func new** in the command prompt and

* It will give you a list of triggers we can select httptrigger for our api functions
* Now it prompts for function name (On PaymentReceived)
* The function got created will have
  + FunctionName as its attribute ex [FunctionName(“OnPaymentReceived”)]
  + HttpTrigger(AuthorizationLevel.Function, Get, Post and Route as its RUN Method parameter arrtibure. Here AuthorizationLevel tells us we need some secret code to run this in the azure after we deploy.
  + HttpRequest as its input parameter.
  + The default route is api/functionname?parameter but we can change this using route parameter in the Trigger attribute
* To run the Azure function locally we can click Debug -> Run or command line “**func host start**”
  + - Important : TO run Azure function locally we need Azure storage emulator install locally and running
  + We can Past the Url it displays on the run command window to a browser or we can Run the following in Powershell command
    - iwr -Method POST -Uri http://localhost:7071/api/OnPaymentReceived -Header @{"Content-Type" = "application/json"} -Body '{}'
    - Or can use Post Man
* Triggers are what causes a function to run. A trigger defines how a function is invoked and function must have exactly one trigger. Triggers have associating data, which is often provided as the payload of the function.
* Along with Triggers Azure function also have **bindings. Binding to a function is a way of declaratively connecting another resource to the function; bindings may be connected as input binding, output binding or both. Data from binding is provided to the function as parameters.**
* **Example of trigger and bindings**

| **Example scenario** | **Trigger** | **Input binding** | **Output binding** |
| --- | --- | --- | --- |
| A new queue message arrives which runs a function to write to another queue. | Queue\* | *None* | Queue\* |
| A scheduled job reads Blob Storage contents and creates a new Cosmos DB document. | Timer | Blob Storage | Cosmos DB |
| The Event Grid is used to read an image from Blob Storage and a document from Cosmos DB to send an email. | Event Grid | Blob Storage and Cosmos DB | SendGrid |
| A webhook that uses Microsoft Graph to update an Excel sheet. | HTTP | *None* | Microsoft Graph |

For using storage binding we need to refer the following Nuget Package

Install-Package Microsoft.Azure.WebJobs.Extensions.Storage -Version 3.0.8

Using

[Queue("order")] IAsyncCollector<Order> orderQueue

And

await orderQueue.AddAsync(order);

we can store our object to a Queue. The Azure storage account should be specified in the setting.json.

1. Go to your Azure portal
2. Select your Storage account
3. In access Keys portion copy the connectionstring
4. Past it in local.setting.json’s AzureWebJobsStorage parameter
5. Once we past the Azure storage connection string .gitignore file includes this connection string so that this connection string will not checked into git.
6. Now if you run the function and call the function using Browser, command line or postman the message will be place in the storage account under Queue.

Complete Script :

namespace AzureFunctions

{

    public static class OnPaymentReceived

    {

        [FunctionName("OnPaymentReceived")]

        public static async Task<IActionResult> Run(

            [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = null)] HttpRequest req,

            [Queue("order")] IAsyncCollector<Order> orderQueue,

            ILogger log)

        {

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

            dynamic order = JsonConvert.DeserializeObject<Order>(requestBody);

            await orderQueue.AddAsync(order);

            log.LogInformation($"Thank you for your paymant. Order Recevied from {order.Email}");

            return new OkObjectResult($"Thank you for your putrchase");

        }

    }

public class Order

{

    public string OrderId { get; set; }

    public string ProductId { get; set;}

    public string Email { get; set; }

    public decimal Price { get; set; }

}

}