Azure Functions

Azure Functions is a serverless solution that allows you to write less code, maintain less infrastructure, and save on costs.

Functions provide a comprehensive set of event-driven triggers and bindings that connect your functions to other services without having to write extra code.

Functions provides native support for developing in C#, Java, JavaScript, PowerShell, Python, plus the ability to use more languages, such as Rust and Go.

Functions provides a variety of hosting options for your business needs and application workload. Event-driven scaling hosting options range from fully serverless, where you only pay for execution time (Consumption plan), to always warm instances kept ready for fastest response times (Premium plan).

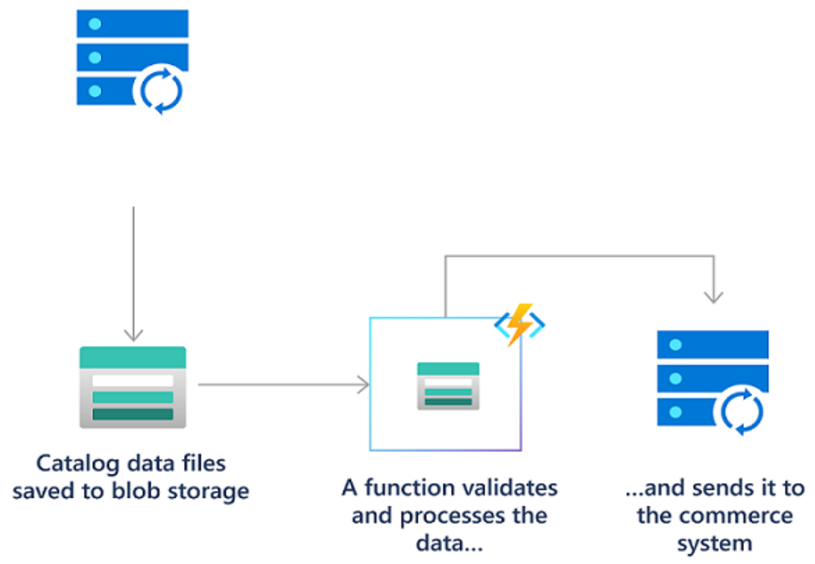
Azure Functions scenarios

Whether you're building a web API, responding to database changes, processing event streams or messages, Azure Functions can be used to implement them.

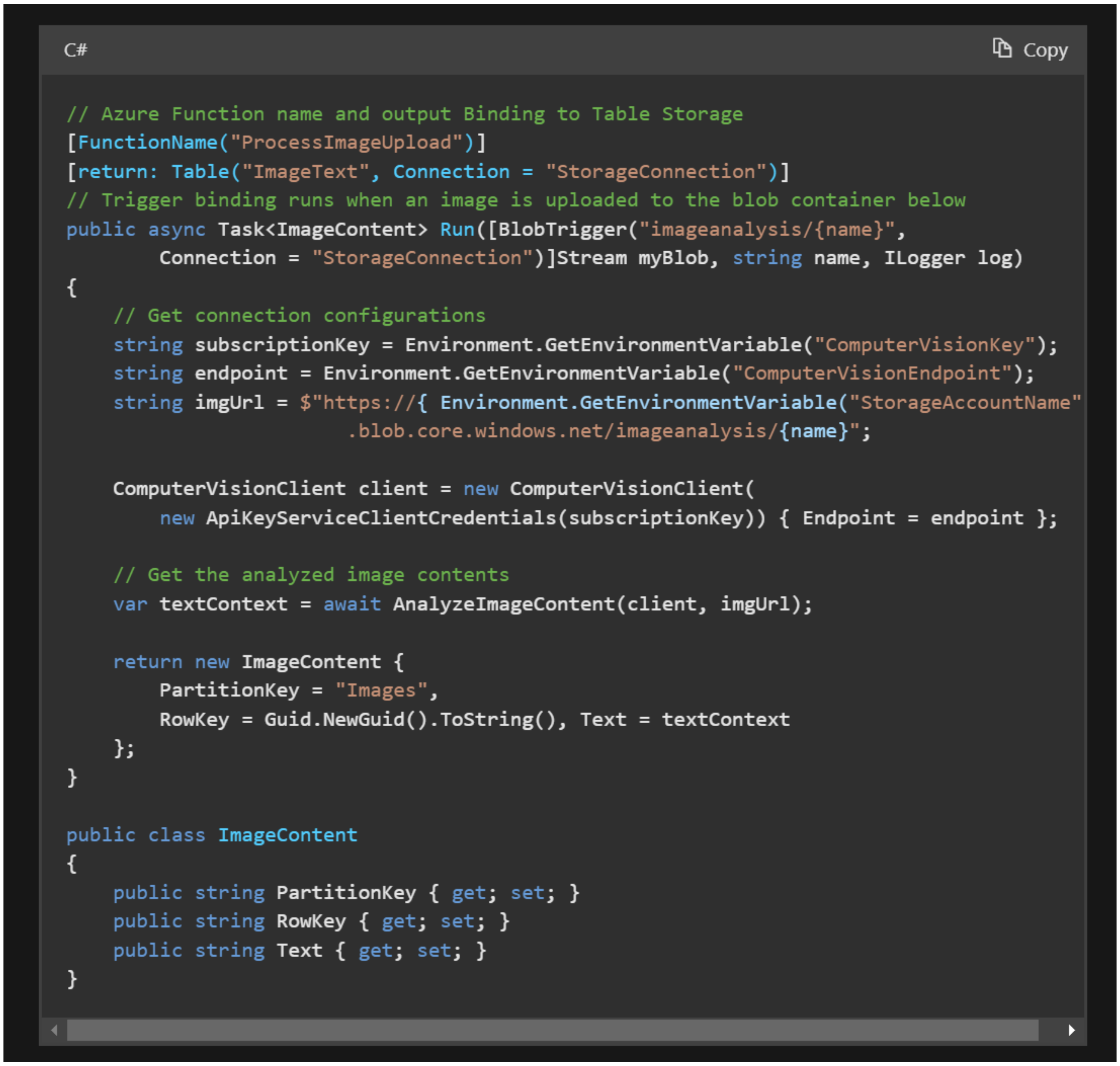
Process file uploads

There are several ways to use functions to process files into or out of a blob storage container.

For example, in a retail solution, a partner system can submit product catalog information as files into blob storage. You can use a blob triggered function to validate, transform, and process the files into the main system as they're uploaded.


Catalog data files saved to blob storage
A function validates and processes the data...
...and sends it to the commerce system

Upload an image to Azure Blob Storage and process it using Azure Functions and Computer Vision. Automate resizing uploaded images using Event Grid. Trigger Azure Functions on blob containers using an event subscription.


C# Copy

```csharp
// Azure Function name and output Binding to Table Storage
[FunctionName("ProcessImageUpload")]
[return: Table("ImageText", Connection = "StorageConnection")]
// Trigger binding runs when an image is uploaded to the blob container below
public async Task<ImageContent> Run([BlobTrigger("imageanalysis/{name}",
        Connection = "StorageConnection")]Stream myBlob, string name, ILogger log)
{
    // Get connection configurations
    string subscriptionKey = Environment.GetEnvironmentVariable("ComputerVisionKey");
    string endpoint = Environment.GetEnvironmentVariable("ComputerVisionEndpoint");
    string imgUrl = $"https://{ Environment.GetEnvironmentVariable("StorageAccountName"
                        .blob.core.windows.net/imageanalysis/{name}";

    ComputerVisionClient client = new ComputerVisionClient(
        new ApiKeyServiceClientCredentials(subscriptionKey)) { Endpoint = endpoint };

    // Get the analyzed image contents
    var textContext = await AnalyzeImageContent(client, imgUrl);

    return new ImageContent {
        PartitionKey = "Images",
        RowKey = Guid.NewGuid().ToString(), Text = textContext
    };
}

public class ImageContent
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Text { get; set; }
}
```

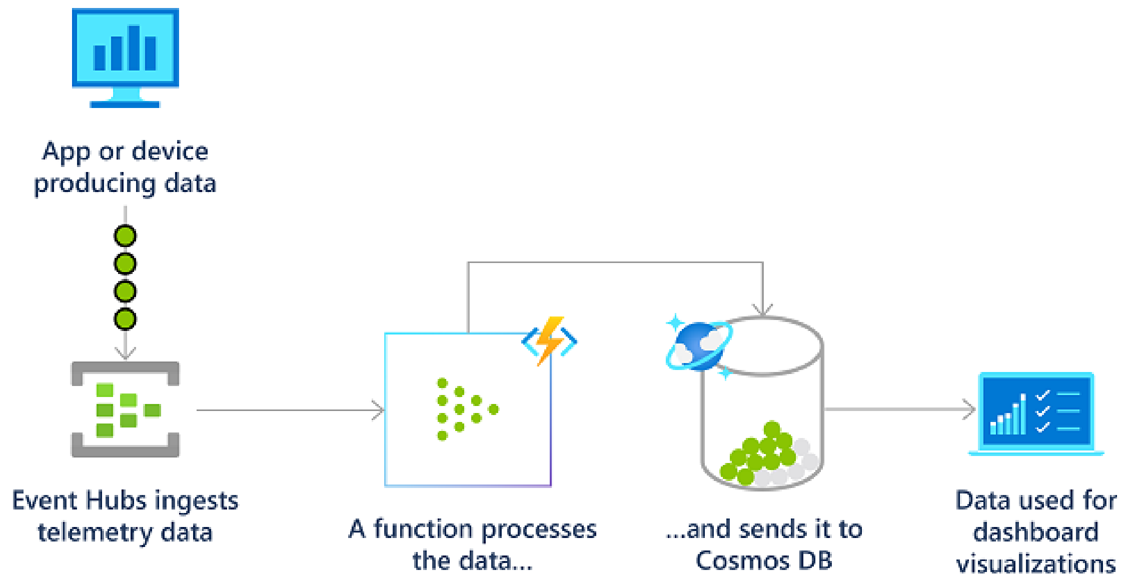
Trigger Azure Functions on blob containers using an event subscription

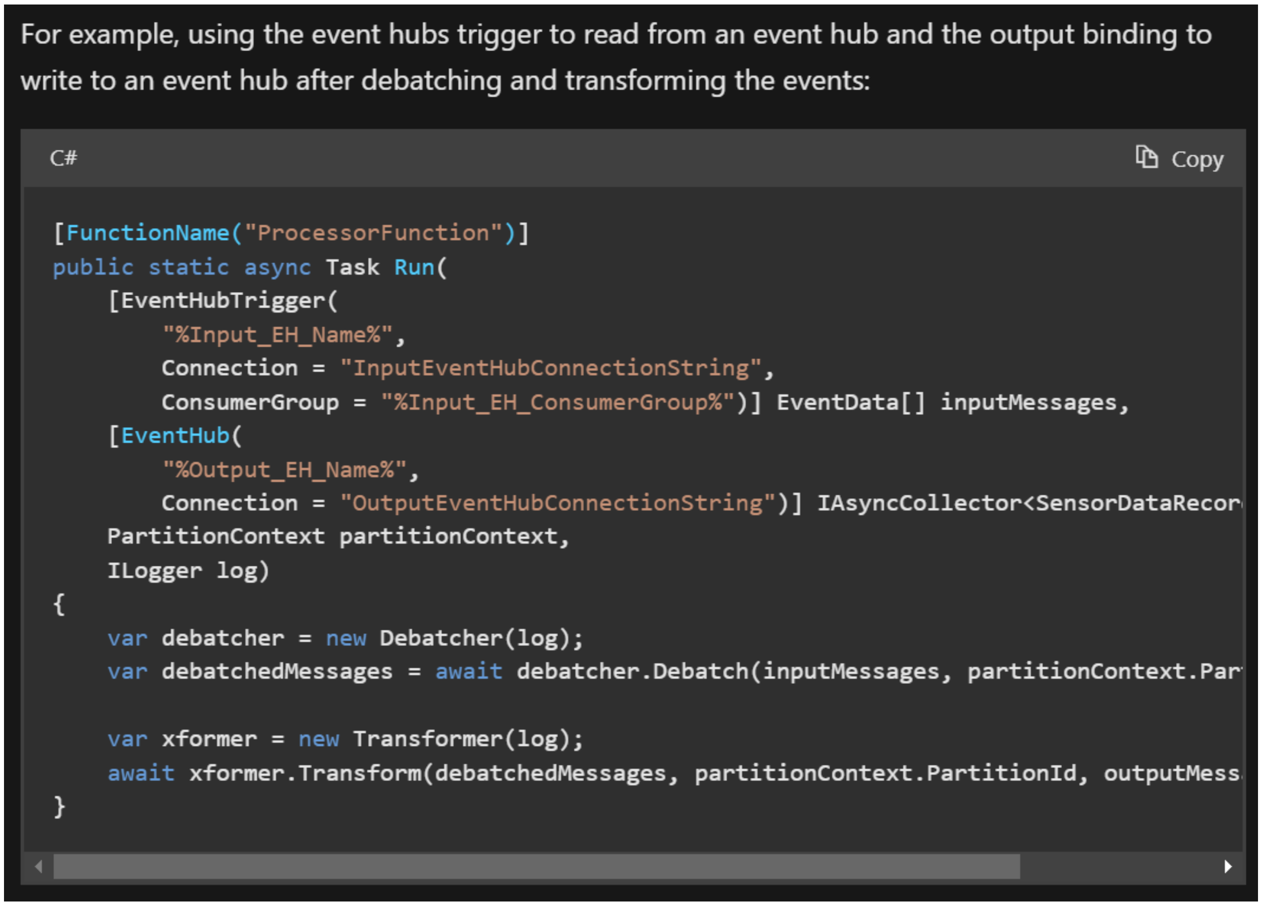
1. Create an event-based Blob Storage triggered function in a new project.
2. Validate locally within Visual Studio Code using the Azurite emulator.
3. Create a blob storage container in a new storage account in Azure.
4. Create a function app in the Flex Consumption plan (preview).
5. Create an event subscription to the new blob container.
6. Deploy and validate your function code in Azure.

Real-time stream and event processing

So much telemetry (process of collecting and analyzing data) is generated and collected from cloud applications, IoT devices, and networking devices. Azure Functions can process that data in near real-time as the hot path, then store it in Azure Cosmos DB for use in an analytics dashboard.

Your functions can also use low-latency event triggers, like Event Grid, and real-time outputs like SignalR to process data in near-real-time.


App or device producing data
Event Hubs ingests telemetry data
A function processes the data...
...and sends it to Cosmos DB
Data used for dashboard visualizations


For example, using the event hubs trigger to read from an event hub and the output binding to write to an event hub after debatching and transforming the events:

C#

Copy

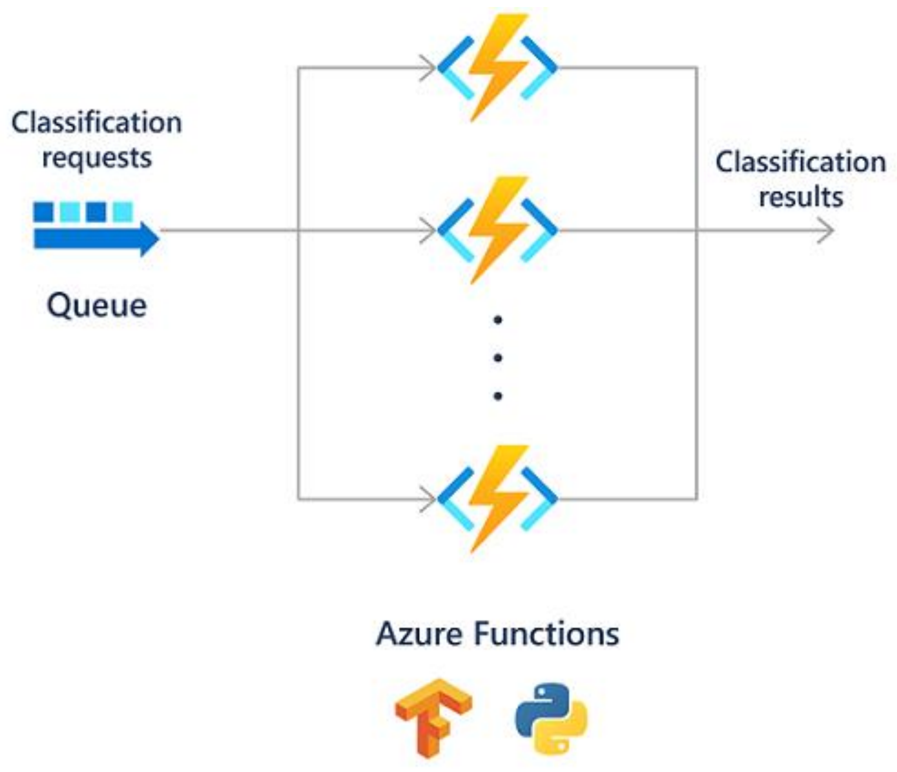
```
[FunctionName("ProcessorFunction")]
public static async Task Run(
    [EventHubTrigger(
        "%Input_EH_Name%",
        Connection = "InputEventHubConnectionString",
        ConsumerGroup = "%Input_EH_ConsumerGroup%")] EventData[] inputMessages,
    [EventHub(
        "%Output_EH_Name%",
        Connection = "OutputEventHubConnectionString")] IAsyncCollector<SensorDataRecor
    PartitionContext partitionContext,
    ILogger log)
{
    var debatcher = new Debatcher(log);
    var debatchedMessages = await debatcher.Debatch(inputMessages, partitionContext.Par

    var xformer = new Transformer(log);
    await xformer.Transform(debatchedMessages, partitionContext.PartitionId, outputMess
}
```

Machine learning and AI

Besides data processing, Azure Functions can be used to infer on models. The Azure OpenAI binding extension lets easily integrate features and behaviors of the Azure OpenAI service into your function code executions.

Functions can connect to OpenAI resources to enable text and chat completions, use assistants, and leverage embeddings and semantic search.

A function might also be called a TensorFlow model or Azure AI services to process and classify a stream of images.

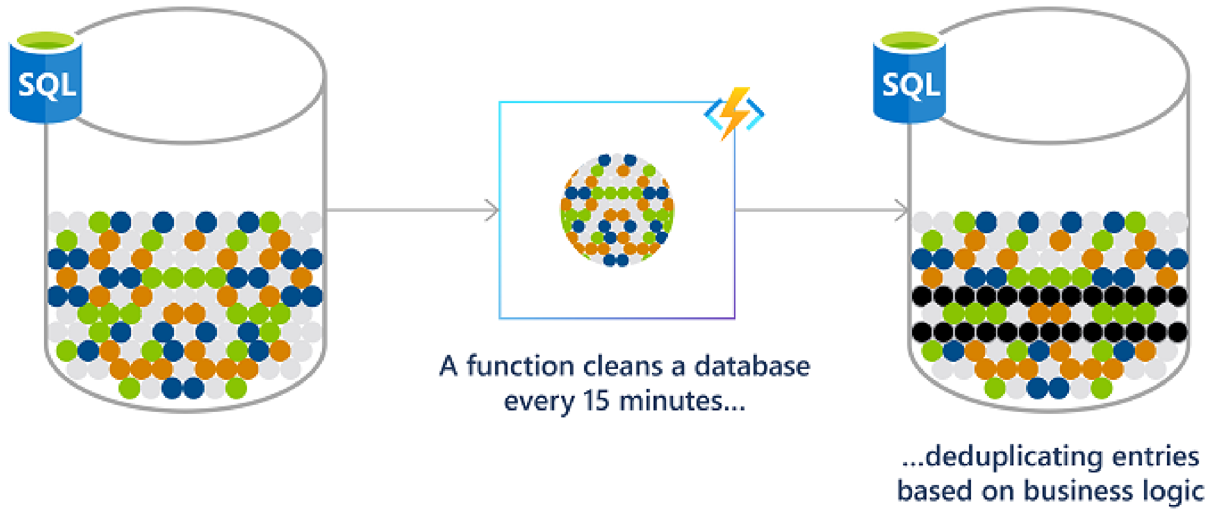


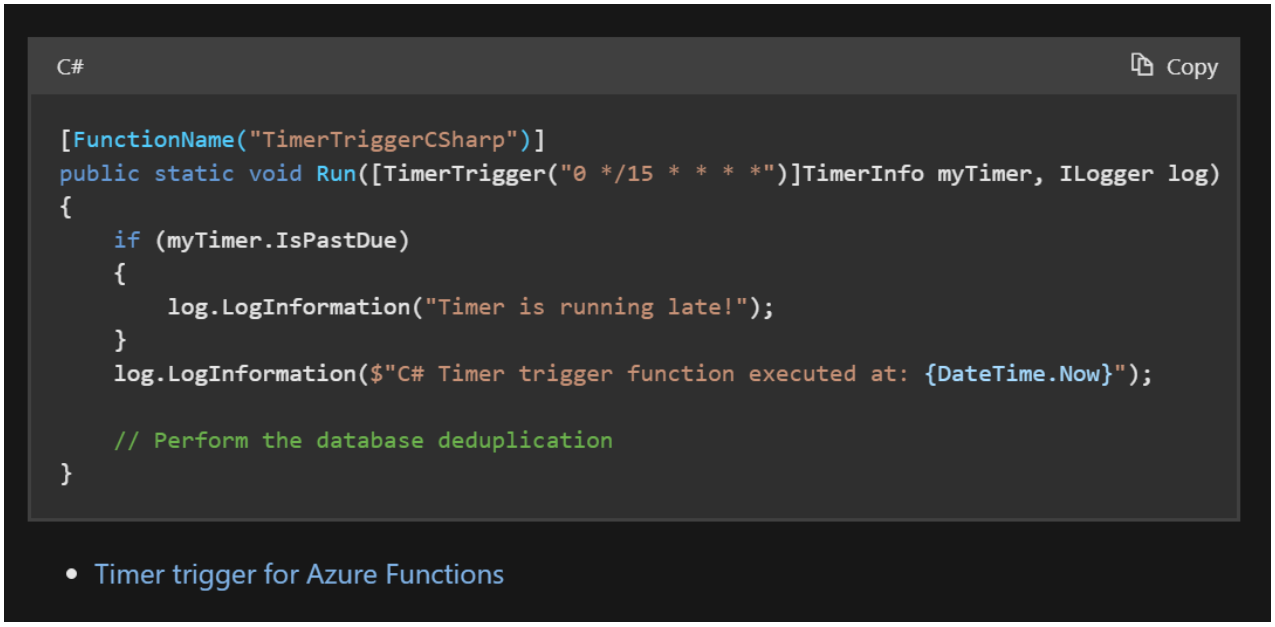
Run scheduled tasks

Functions enables you to run your code based on a cron schedule that you define.

Check out how to Create a function in the Azure portal that runs on a schedule.

A customer financial services database, for example, might be analyzed for duplicate entries every 15 minutes to avoid multiple communications going out to the same customer.


SQL
A function cleans a database every 15 minutes...
SQL
...deduplicating entries based on business logic



C# Copy

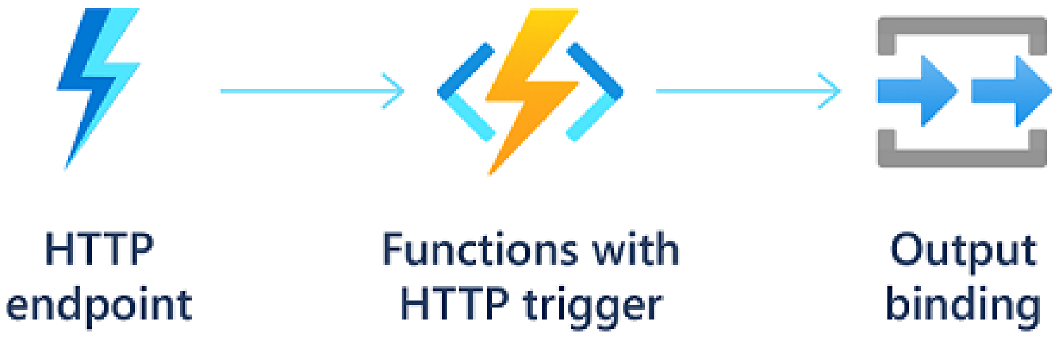
```csharp
[FunctionName("TimerTriggerCSharp")]
public static void Run([TimerTrigger("0 */15 * * * *")]TimerInfo myTimer, ILogger log)
{
    if (myTimer.IsPastDue)
    {
        log.LogInformation("Timer is running late!");
    }
    log.LogInformation($"C# Timer trigger function executed at: {DateTime.Now}");

    // Perform the database deduplication
}
```

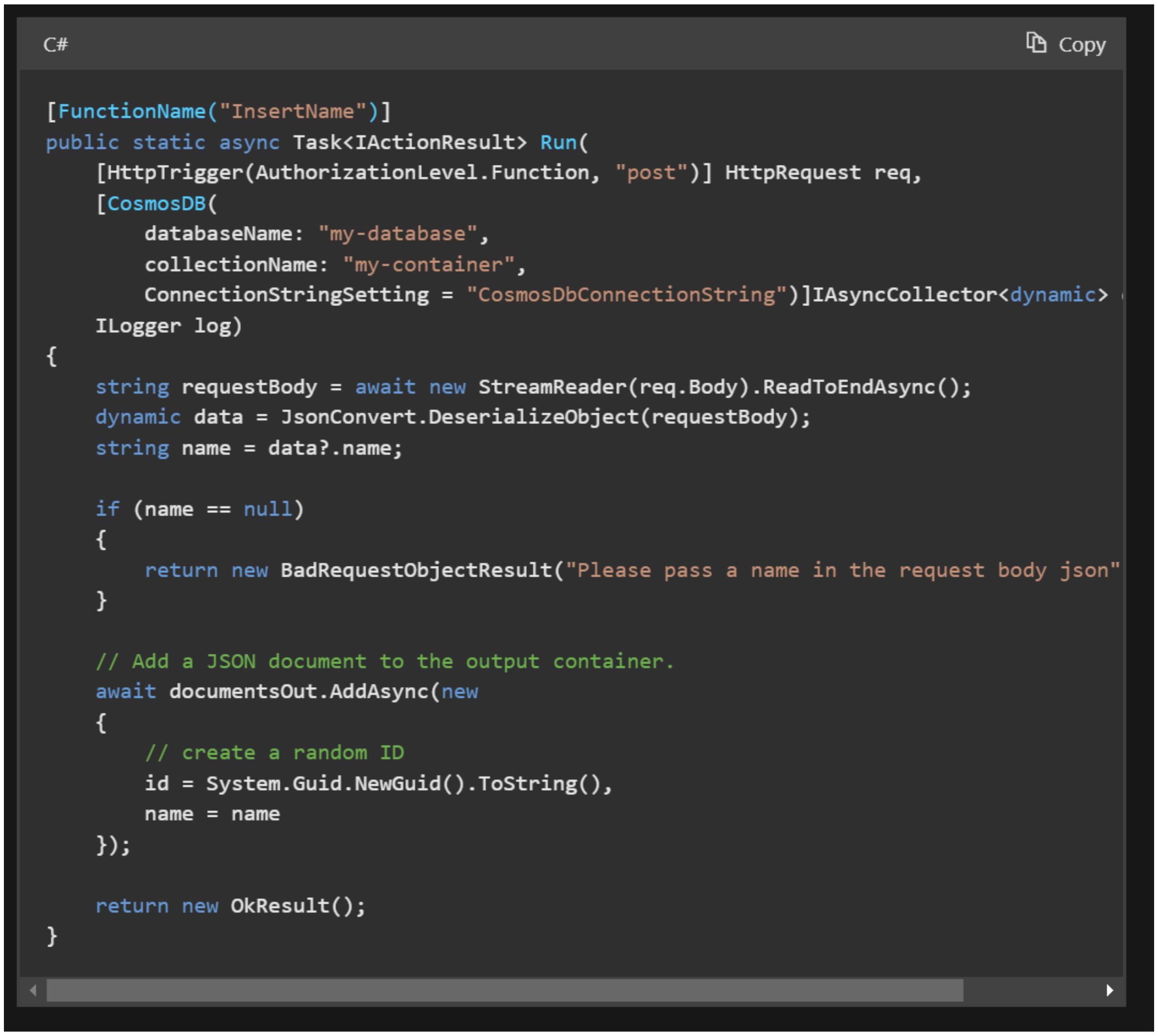
- Timer trigger for Azure Functions

Build a scalable web API

An HTTP triggered function defines an HTTP endpoint. These endpoints run function code that can connect to other services directly or by using binding extensions. You can compose the endpoints into a web-based API.

You can also use an HTTP triggered function endpoint as a webhook integration, such as GitHub webhooks. In this way, you can create functions that process data from GitHub events.

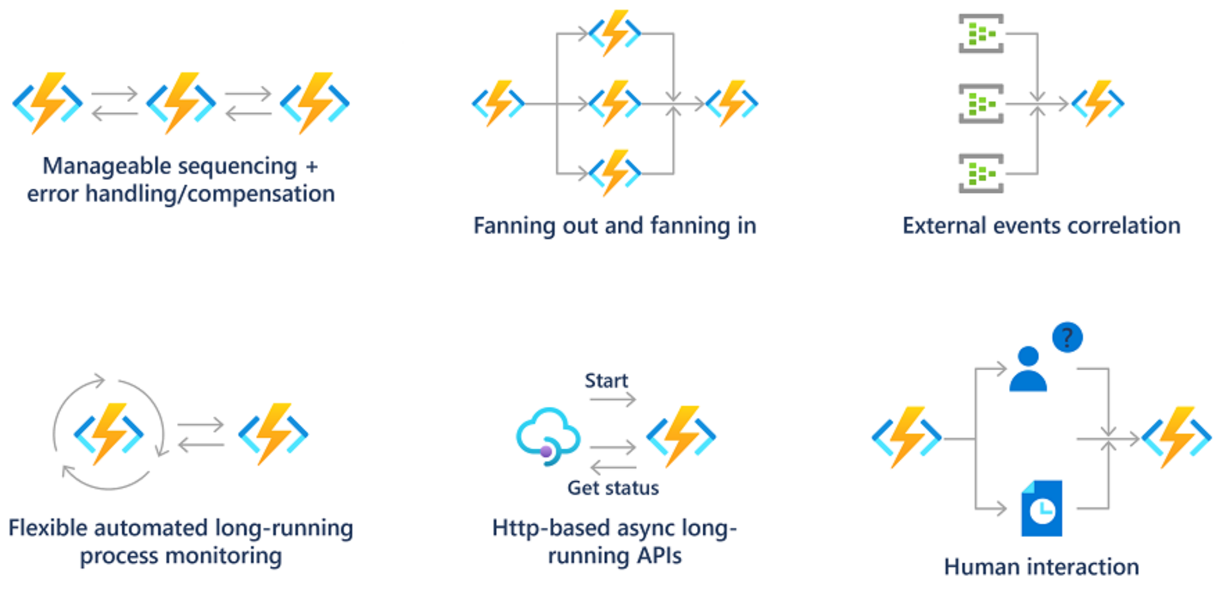




```csharp
[FunctionName("InsertName")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "post")] HttpRequest req,
    [CosmosDB(
        databaseName: "my-database",
        collectionName: "my-container",
        ConnectionStringSetting = "CosmosDbConnectionString")]IAsyncCollector<dynamic>
    ILogger log)
{
    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    string name = data?.name;

    if (name == null)
    {
        return new BadRequestObjectResult("Please pass a name in the request body json"
    }

    // Add a JSON document to the output container.
    await documentsOut.AddAsync(new
    {
        // create a random ID
        id = System.Guid.NewGuid().ToString(),
        name = name
    });

    return new OkResult();
}
```

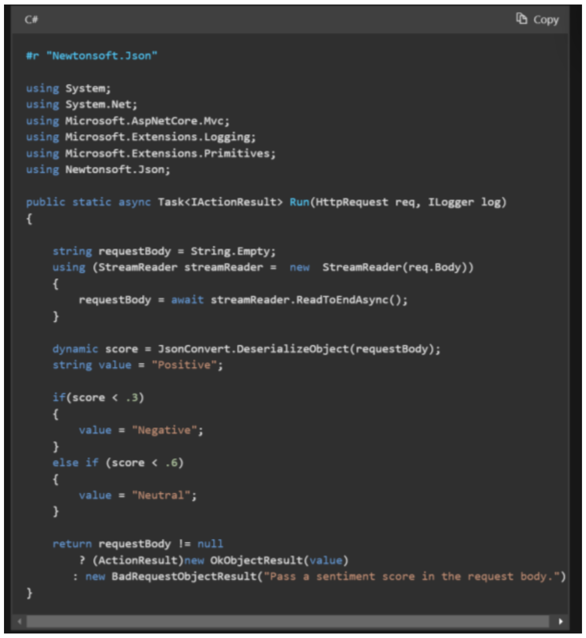
Build a serverless workflow

Functions are often the compute component in a serverless workflow topology, such as a Logic Apps workflow. You can also create long-running orchestrations using the Durable Functions extension. For more information, see Durable Functions overview.


Manageable sequencing + error handling/compensation
Fanning out and fanning in
External events correlation
Flexible automated long-running process monitoring
Start
Get status
Http-based async long-running APIs
Human interaction

Azure Functions integrates with Azure Logic Apps in the Logic Apps Designer. This integration allows you use the computing power of Functions in orchestrations with other Azure and third-party services.

Create an Azure AI services API Resource.Create a function that categorizes tweet sentiment. (Create the function app and Create an HTTP-triggered function)

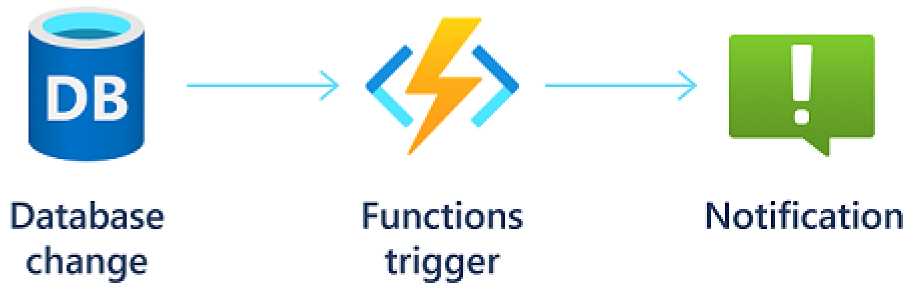


```
#r "Newtonsoft.Json"

using System;
using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;

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

    string requestBody = String.Empty;
    using (StreamReader streamReader =  new  StreamReader(req.Body))
    {
        requestBody = await streamReader.ReadToEndAsync();
    }

    dynamic score = JsonConvert.DeserializeObject(requestBody);
    string value = "Positive";

    if(score < .3)
    {
        value = "Negative";
    }
    else if (score < .6)
    {
        value = "Neutral";
    }

    return requestBody != null
        ? (ActionResult)new OkObjectResult(value)
       : new BadRequestObjectResult("Pass a sentiment score in the request body.")
}
```

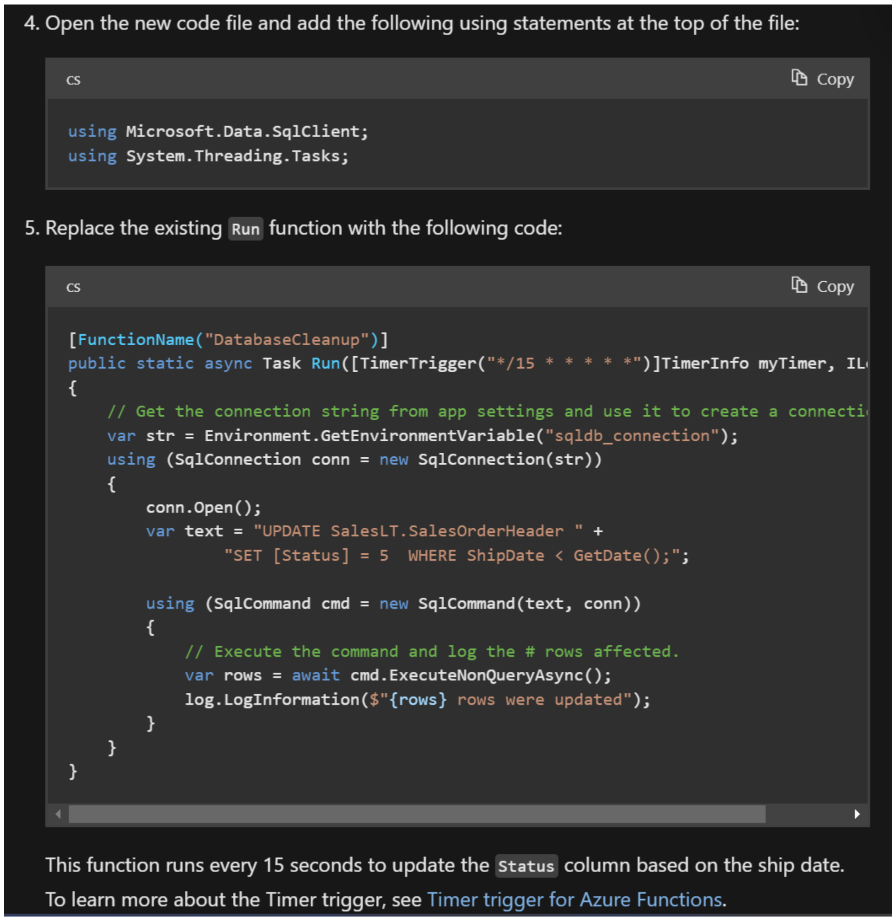
Create a logic app that connects to X. Add sentiment detection to the logic app. Connect the logic app to the function. Send an email based on the response from the function.

Respond to database changes

There are processes where you might need to log, audit, or perform some other operation when stored data changes. Functions triggers provide a good way to get notified of data changes to initial such an operation.



Azure Functions to create a scheduled job that connects to an Azure SQL Database or Azure SQL Managed Instance. The function code cleans up rows in a table in the database.


4. Open the new code file and add the following using statements at the top of the file:

```cs
using Microsoft.Data.SqlClient;
using System.Threading.Tasks;
```

5. Replace the existing `Run` function with the following code:

```cs
[FunctionName("DatabaseCleanup")]
public static async Task Run([TimerTrigger("*/15 * * * * *")]TimerInfo myTimer, IL
{
    // Get the connection string from app settings and use it to create a connecti
    var str = Environment.GetEnvironmentVariable("sqldb_connection");
    using (SqlConnection conn = new SqlConnection(str))
    {
        conn.Open();
        var text = "UPDATE SalesLT.SalesOrderHeader " +
                "SET [Status] = 5  WHERE ShipDate < GetDate();";

        using (SqlCommand cmd = new SqlCommand(text, conn))
        {
            // Execute the command and log the # rows affected.
            var rows = await cmd.ExecuteNonQueryAsync();
            log.LogInformation($"{rows} rows were updated");
        }
    }
}
```

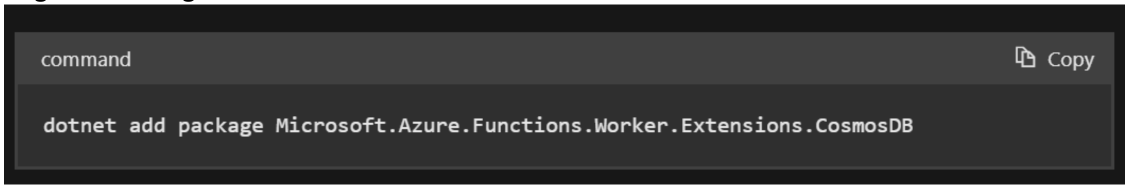
This function runs every 15 seconds to update the `Status` column based on the ship date.

To learn more about the Timer trigger, see Timer trigger for Azure Functions.

Azure Functions lets you connect Azure services and other resources to functions without having to write your own integration code. These *bindings*, which represent both input and output, are declared within the function definition. Data from bindings is provided to the function as parameters. A *trigger* is a special type of input binding.

Although a function has only one trigger, it can have multiple input and output bindings. The output binding that you add to this function writes data from the HTTP request to a JSON document stored in an Azure Cosmos DB container.

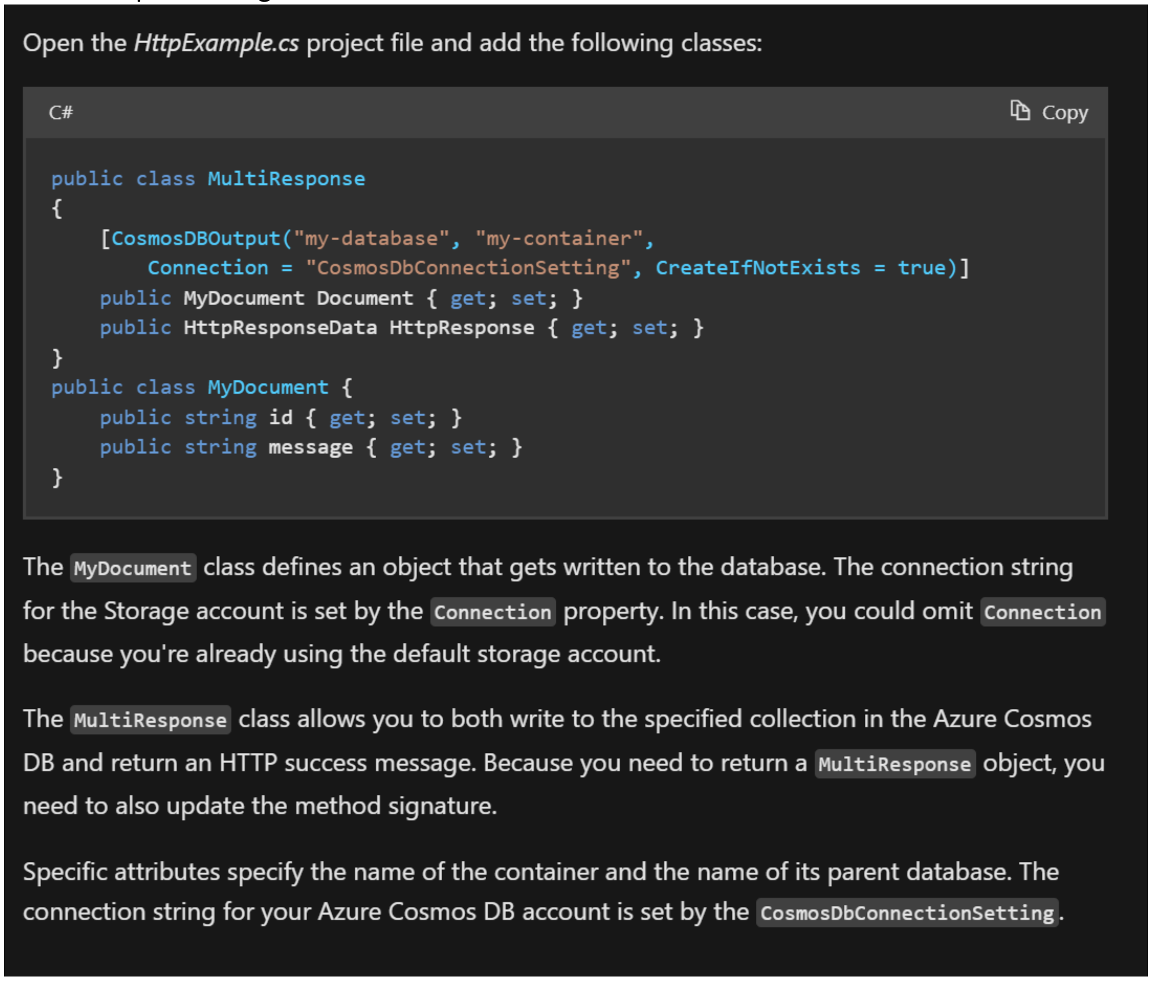
1. Configure your local environment
2. Create your Azure Cosmos DB account
3. Configure your local environment
4. Create an Azure Cosmos DB database and container
5. Update your function app settings
6. Register binding extensions



command

```
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.CosmosDB
```

7. Add an output binding



Open the *HttpExample.cs* project file and add the following classes:

C#

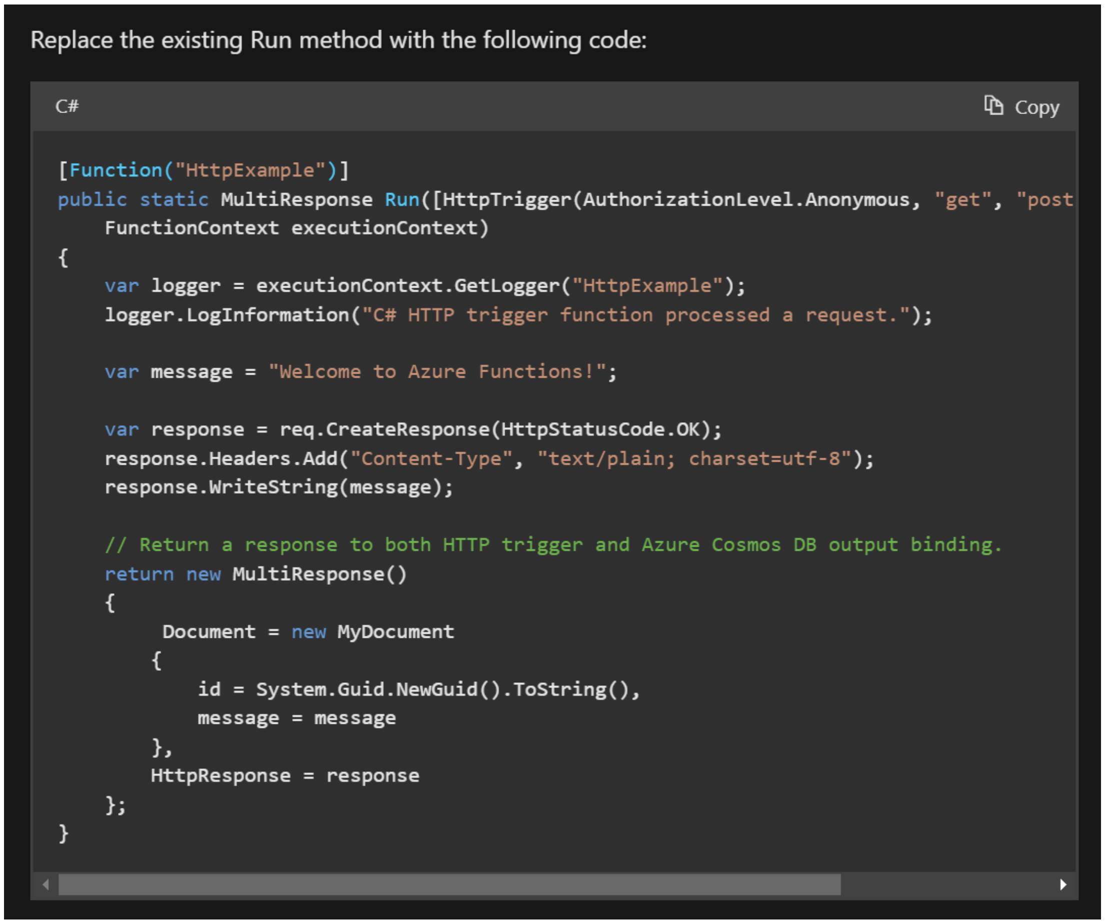
```csharp
public class MultiResponse
{
    [CosmosDBOutput("my-database", "my-container",
        Connection = "CosmosDbConnectionSetting", CreateIfNotExists = true)]
    public MyDocument Document { get; set; }
    public HttpResponseData HttpResponse { get; set; }
}
public class MyDocument {
    public string id { get; set; }
    public string message { get; set; }
}
```

The `MyDocument` class defines an object that gets written to the database. The connection string for the Storage account is set by the `Connection` property. In this case, you could omit `Connection` because you're already using the default storage account.

The `MultiResponse` class allows you to both write to the specified collection in the Azure Cosmos DB and return an HTTP success message. Because you need to return a `MultiResponse` object, you need to also update the method signature.

Specific attributes specify the name of the container and the name of its parent database. The connection string for your Azure Cosmos DB account is set by the `CosmosDbConnectionSetting`.

8. Add code that uses the output binding



Replace the existing Run method with the following code:

C#

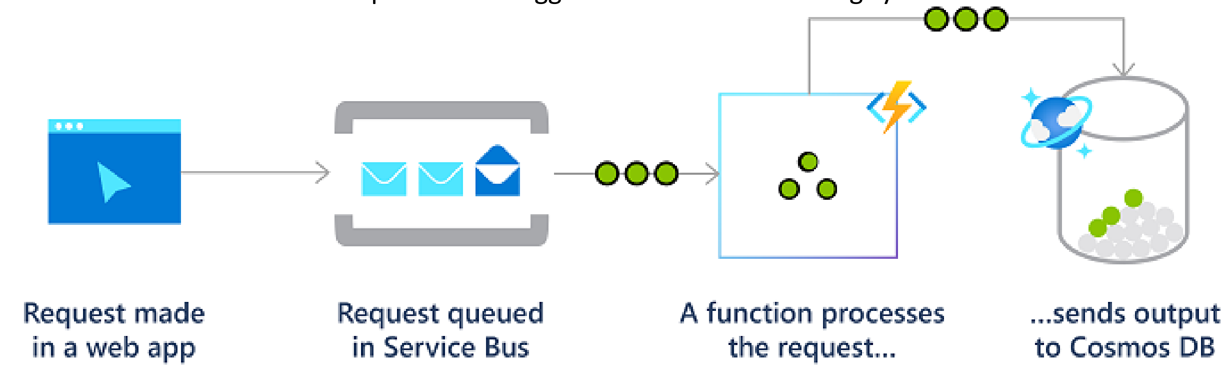
Copy

```
[Function("HttpExample")]
public static MultiResponse Run([HttpTrigger(AuthorizationLevel.Anonymous, "get", "post
    FunctionContext executionContext)
{
    var logger = executionContext.GetLogger("HttpExample");
    logger.LogInformation("C# HTTP trigger function processed a request.");

    var message = "Welcome to Azure Functions!";

    var response = req.CreateResponse(HttpStatusCode.OK);
    response.Headers.Add("Content-Type", "text/plain; charset=utf-8");
    response.WriteString(message);

    // Return a response to both HTTP trigger and Azure Cosmos DB output binding.
    return new MultiResponse()
    {
         Document = new MyDocument
        {
            id = System.Guid.NewGuid().ToString(),
            message = message
        },
        HttpResponse = response
    };
}
```

Create reliable message systems

You can use Functions with Azure messaging services to create advanced event-driven messaging solutions.

For example, you can use triggers on Azure Storage queues to chain together a series of function executions. Or use service bus queues and triggers for an online ordering system.


Request made in a web app
Request queued in Service Bus
A function processes the request...
...sends output to Cosmos DB

Connect Azure Functions to Azure Storage using Visual Studio Code

1. Configure your local environment

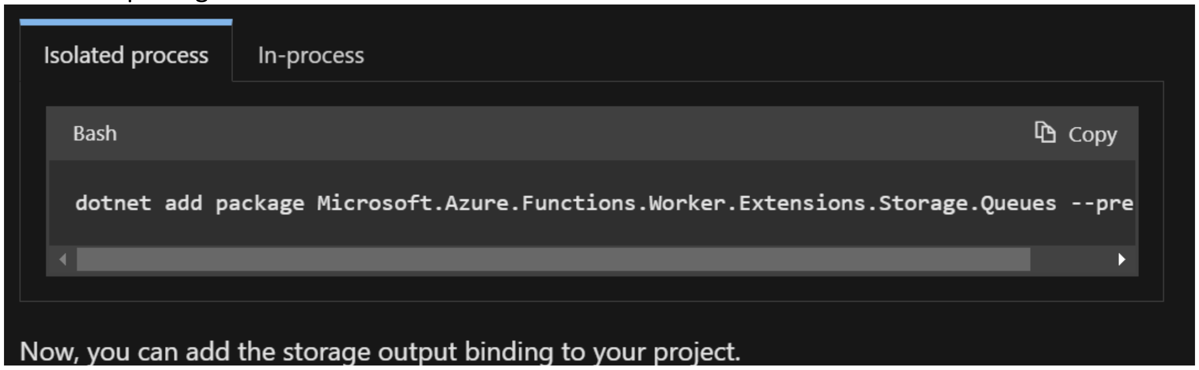
Install the Azure Storage extension for Visual Studio Code. Install Azure Storage Explorer. Install .NET Core CLI tools.

Storage Explorer is a tool that you'll use to examine queue messages generated by your output binding. Storage Explorer is supported on macOS, Windows, and Linux-based operating systems.

2. Download the function app settings

The connection string for this account is stored securely in the app settings in Azure.To connect to your storage account when running the function locally, you must download app settings to the *local.settings.json* file.

3. Register binding extensions - Except for HTTP and timer triggers, bindings are implemented as extension packages



Isolated process | In-process

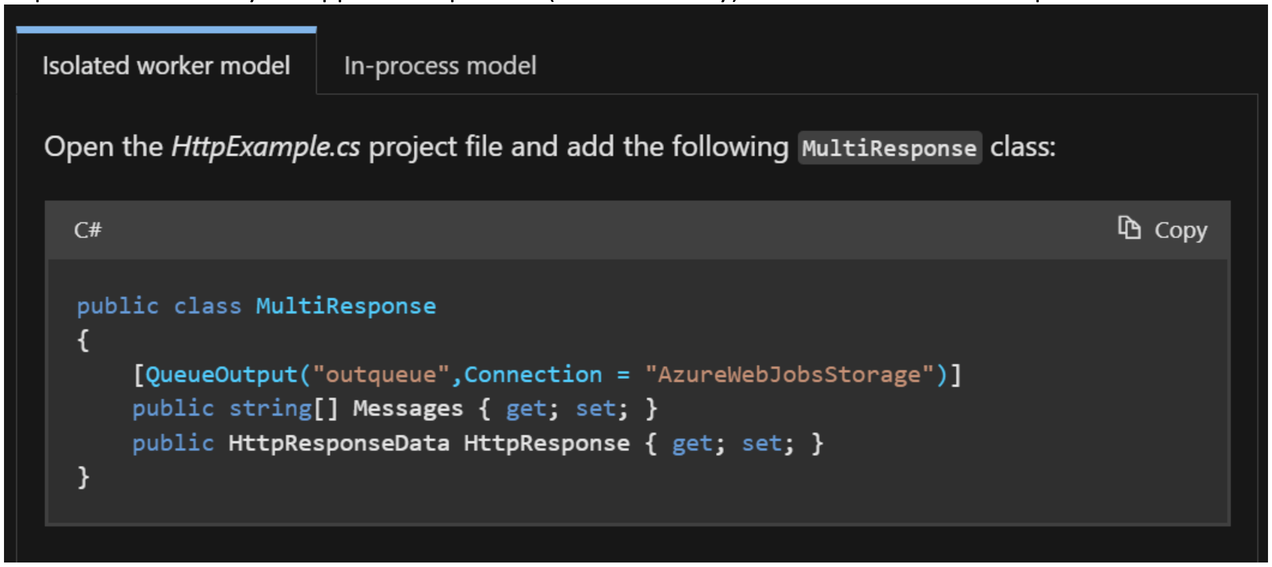
Bash

```bash
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues --pre
```

Now, you can add the storage output binding to your project.

4. Add an output binding

In a C# project, the bindings are defined as binding attributes on the function method. Specific definitions depend on whether your app runs in-process (C# class library) or in an isolated worker process.



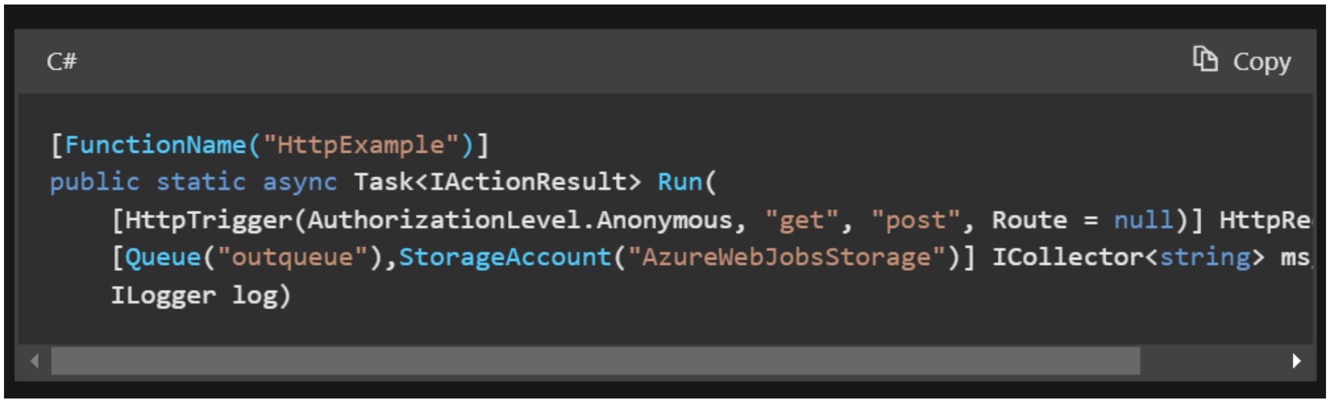
Isolated worker model | In-process model

Open the *HttpExample.cs* project file and add the following `MultiResponse` class:

C#

```csharp
public class MultiResponse
{
    [QueueOutput("outqueue",Connection = "AzureWebJobsStorage")]
    public string[] Messages { get; set; }
    public HttpResponseData HttpResponse { get; set; }
}
```

In process model,

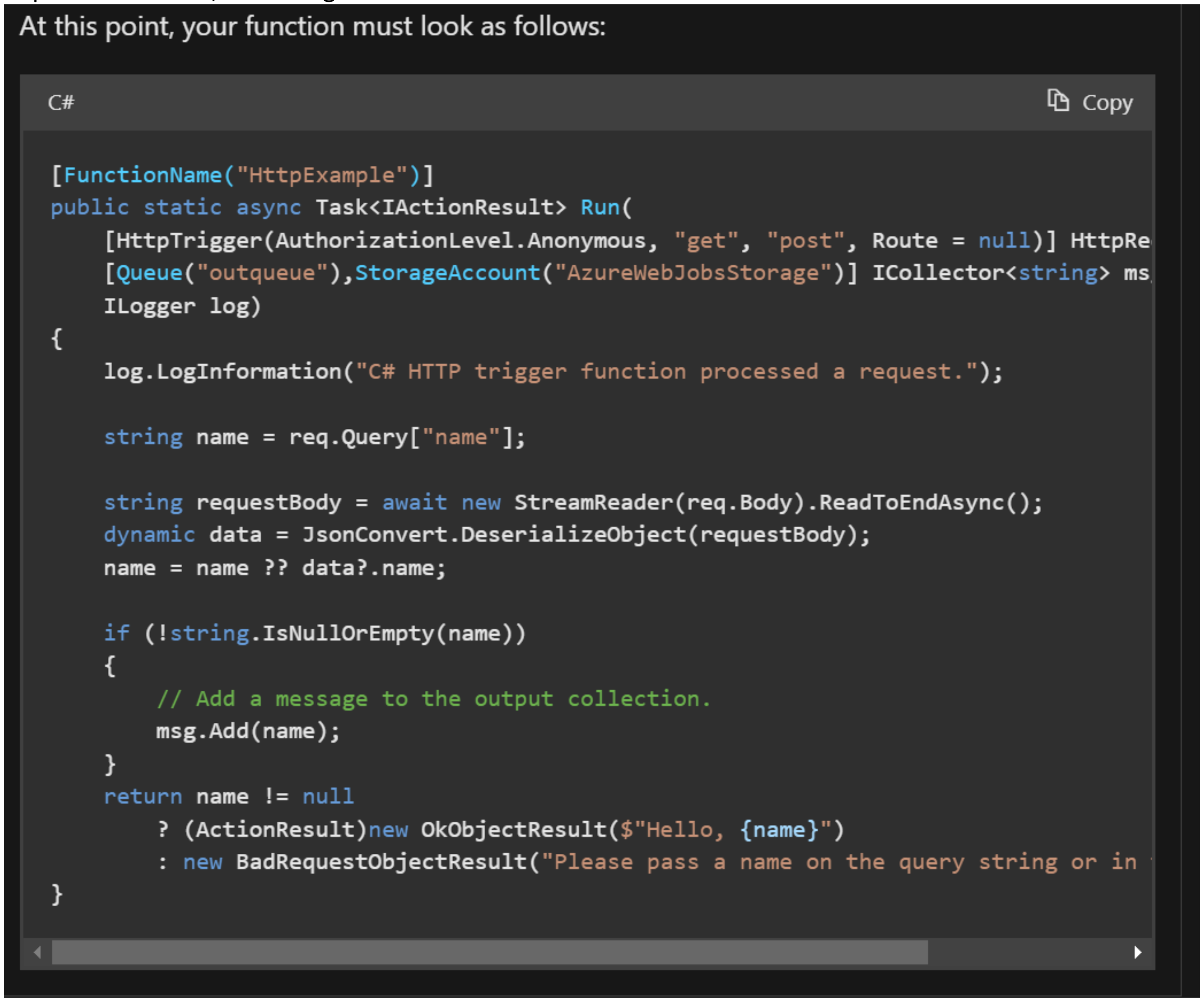

C#

Copy

```csharp
[FunctionName("HttpExample")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)] HttpRe
    [Queue("outqueue"),StorageAccount("AzureWebJobsStorage")] ICollector<string> ms
    ILogger log)
```

5. Add code that uses the output binding

By using an output binding, you don't have to use the Azure Storage SDK code for authentication, getting a queue reference, or writing data.


At this point, your function must look as follows:

C#

Copy

```csharp
[FunctionName("HttpExample")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)] HttpRe
    [Queue("outqueue"),StorageAccount("AzureWebJobsStorage")] ICollector<string> ms
    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;

    if (!string.IsNullOrEmpty(name))
    {
        // Add a message to the output collection.
        msg.Add(name);
    }
    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query string or in
}
```

6. Run the function locally
7. Connect Storage Explorer to your account

Create a function triggered by Azure Queue storage

1. Create a function triggered by Azure Queue storage
2. Create an Azure Function app - Under Select a hosting option, select Consumption > Select to create your app in the default Consumption plan
3. Create a Queue triggered function - Under Select a template, scroll down and choose the Azure Queue Storage trigger template.
4. Create the queue - In the storage account page, select Data storage > Queues > + Queue. {myqueue-items}

Azure Functions Consumption plan hosting

When you're using the Consumption plan, instances of the Azure Functions host are dynamically added and removed based on the number of incoming events. The Consumption plan, along with the Flex Consumption plan, is a fully serverless hosting option for Azure Functions.

The Consumption plan scales automatically, even during periods of high load.
When running functions in a Consumption plan, you're charged for compute resources only when your functions are running.
On a Consumption plan, a function execution times out after a configurable period of time.

Azure Functions Flex Consumption plan hosting

Flex Consumption is a Linux-based Azure Functions hosting plan that builds on the Consumption *pay for what you use* serverless billing model.
It gives you more flexibility and customizability by introducing private networking, instance memory size selection, and fast/large scale-out features still based on a *serverless* model.

- The Flex Consumption plan is currently in preview.
- Always-ready instances
- Virtual network integration
- Fast scaling based on concurrency for both HTTP and non-HTTP apps
- Multiple choices for instance memory sizes

Billing is based on number of executions, execution time, and memory used. Usage is aggregated across all functions within a function app.

Azure Functions Premium plan hosting

The Azure Functions Elastic Premium plan is a dynamic scale hosting option for function apps.

- Avoid cold starts with warm instances.
- Virtual network connectivity.
- Supports longer runtime durations.
- Choice of Premium instance sizes.
- More predictable pricing, compared with the Consumption plan.
- High-density app allocation for plans with multiple function apps.
- Supports Linux container deployments.

When you're using the Premium plan, instances of the Azure Functions host are added and removed based on the number of incoming events, just like the Consumption plan.

Billing for the Premium plan is based on the number of core seconds and memory allocated across instances.
Multiple function apps can be deployed to the same Premium plan, and the plan allows you to configure compute instance size, base plan size, and maximum plan size.

Dedicated hosting plans for Azure Functions

Hosting your function app with dedicated resources in an App Service plan, including in an App Service Environment (ASE).
An App Service plan defines a set of dedicated compute resources for an app to run. These dedicated compute resources are analogous to the server farm in conventional hosting.

One or more function apps can be configured to run on the same computing resources (App Service plan) as other App Service apps, such as web apps.

The dedicated App Service plans supported for function app hosting include Basic, Standard, Premium, and Isolated SKUs.

Important

Free and Shared tier App Service plans aren't supported by Azure Functions.
For a lower-cost option hosting your function executions, you should instead consider the Consumption plan or the Flex Consumption plan, where you are billed based on function executions.

Consider a dedicated App Service plan in the following situations:

- You have existing, underutilized VMs that are already running other App Service instances.
- You want to provide a custom image on which to run your functions.

You pay for function apps in an App Service Plan as you would for other App Service resources.