# Create Azure Functions

## Serverless Compute

### What is serverless compute?

[Serverless compute](https://azure.microsoft.com/solutions/serverless/) can be thought of as a function as a service (FaaS), or a microservice that is hosted on a cloud platform. Your business logic runs as functions, and you don't have to manually provision or scale infrastructure. Your app is automatically scaled out or down depending on load.

Azure has several ways to build this sort of architecture. The two most common approaches are Azure Logic Apps and Azure Functions, which we'll focus on in this module.

### What is Azure Functions?

Azure Functions is a serverless application platform. It enables developers to host business logic that can be executed without provisioning infrastructure. Functions provides intrinsic scalability and you are charged only for the resources used.

You can write your function code in the language of your choice, including C#, F#, JavaScript, Python, and PowerShell Core. Support for package managers like NuGet and NPM is also included.

### Characteristics of a serverless solution

1. Avoids over-allocation of infrastructure
2. Stateless logic

– If state is required, it can be stored in an associated storage service.

1. Event driven

– Run only in response to an event called a "trigger".

1. Can be used in traditional compute environments

– Should the needs of your app change, you can take your project and deploy it in a non-serverless environment, which gives you the flexibility

### Drawbacks of a serverless solution

1. Execution time

– Default is 5 mins, max of 10 minutes.

– If function requires more than 10 minutes to execute, you can host it on a VM.

– For HTTP request and an HTTP response, the timeout is further restricted to 2.5 minutes.

– Check [**Durable Functions**](https://docs.microsoft.com/en-us/azure/azure-functions/durable) - enables you to orchestrate the executions of multiple functions without any timeout.

1. Execution frequency

– If you expect your function to be executed continuously by multiple clients, it would be prudent to estimate the usage and calculate the cost of using functions accordingly. It might be cheaper to host your service on a VM.

## Create a function app in the Azure portal

### Choose a service plan

* 1. Consumption plan:
     + With timeout
     + Automatic scaling
     + Bills you only when your functions are running
  2. Azure App Service plan:
     + Run continuously on a VM that you define
* Not a serverless plan technically

### Storage account requirements

A function app must be linked to a storage account. You can select an existing account or create a new one.

The function app uses this storage account for internal operations, such as logging function executions and managing execution triggers.

On the Consumption plan, this is also where the function code and configuration file are stored.

### Create a function app

1. Sign in to the [Azure portal](https://portal.azure.com/learn.docs.microsoft.com) using the same account you used to activate the sandbox.
2. Under Azure services, select Create a resource. The Create a resource pane appears.
3. In the menu, select Compute, and then select Function App in the Popular products list. The Create Function App pane appears.
4. On the Basics tab, enter the values for each setting – refer next page.
5. Select Review + create, and then select Create. Deployment will take a few minutes. You'll receive a notification when deployment is completed.

### Verify your Azure function app

1. When deployment completes, select Go to resource. The Function App pane for your escalator function appears.
2. In the Essentials section, select the URL link to open it in a browser. A default Azure web page appears with a message that your Functions app is up and running.

| **Setting** | | **Value** |
| --- | --- | --- |
| **Project Details** | Subscription | Concierge Subscription |
| Resource Group | From the dropdown list, select your resource group  **learn-042c9dc4-92bf-40c0-ab63-2283ea7327cd** |
| **Instance Details** | Function App name | Enter a globally unique app name, which becomes part of the base URL of your service. For example, you can name it **escalator-functions-xxx**, where xxx can be replaced with your initials and a number. Valid characters are a-z, 0-9 and - |
| Publish | Code |
| Runtime stack | Node.js (which is the language we use to implement the function examples in this exercise). |
| Version | Accept *default* |
| Region | Select a geographical location close to you. In a production system, you would want to select a location near your customers or consumers of the function. |

## Create a function in your function app

Create function -> Add function

## Test your Azure function

### Run function manually

select a function that you created in your function app

-> Code + Test -> (Get function URL)

### Test in the Azure portal

select a function that you created in your function app

-> Code + Test -> Test/Run -> Input -> Run -> Output

[Click here | Exercise - Add logic to the function app - Learn | Microsoft Docs](https://docs.microsoft.com/en-us/learn/modules/create-serverless-logic-with-azure-functions/5-add-logic-to-the-function-app?pivots=javascript)

## Monitoring and Application Insights dashboard

In the ‘Function App’ menu, under ‘**Settings’**, select ‘**Application Insights’**

**-> Turn on Application Insights** -> **Apply** -> **Yes.**

## Run your code on-demand with Azure Functions

### Bindings:

A binding is a declarative way to connect data and services to your function.

Bindings interact with various data sources, which means you don't have to write the code in your function to connect to data sources and manage connections--the platform takes care of that complexity for you as part of the binding code. Each binding has a direction -- your code reads data from input bindings and writes data to output bindings. Each function can have zero or more bindings to manage the input and output data processed by the function.

A trigger is a type of input binding that has the ability to initiate execution of some code.

Azure provides a large number of bindings to connect to different storage and messaging services.

### Triggers:

Functions are event driven.

The type of event that starts a function is called a **trigger**.

Each function must be configured with exactly one trigger.

Azure supports triggers for the following services.

| **Service** | **Trigger description** |
| --- | --- |
| Blob Storage | Starts a function when a new or updated blob is detected. |
| Azure Cosmos DB | Start a function when inserts and updates are detected. |
| Event Grid | Starts a function when an event is received from Event Grid. |
| HTTP | Starts a function with an HTTP request. |
| Microsoft Graph Events | Starts a function in response to an incoming webhook from the Microsoft Graph. Each instance of this trigger can react to one Microsoft Graph resource type. |
| Queue Storage | Starts a function when a new item is received on a queue. The queue message is provided as input to the function. |
| Service Bus | Starts a function in response to messages from a Service Bus queue. |
| Timer | Starts a function on a schedule. |

### Define a sample binding:

Let's look at an example of configuring a function with an input binding (trigger) and an output binding.

#### Function.json:

#### Example 1: Azure Queue Storage trigger input binding and an Azure Table storage output binding

Let's say we want to write a new row to Azure Table storage whenever a new message appears in Azure Queue Storage.

{

"bindings": [{

"name": "order",

"type": "queueTrigger",

"direction": "in",

"queueName": "myqueue-items",

"connection": "MY\_STORAGE\_ACCT\_APP\_SETTING"

},

{

"name": "$return",

"type": "table",

"direction": "out",

"tableName": "outTable",

"connection": "MY\_TABLE\_STORAGE\_ACCT\_APP\_SETTING"

}

]

}

#### Example 2: Http trigger input binding and http output binding

{

"bindings": [

{

"authLevel": "function",

"type": "httpTrigger",

"direction": "in",

"name": "req",

"methods": [ "get", "post" ]

},

{

"type": "http",

"direction": "out",

"name": "res"

}

]

}

#### Example 3: Azure Queue Storage trigger input binding and an Azure Table storage output binding

Let's say we want to execute an action every 5 mins.

{

"bindings": [

{

"name": "myTimer",

"type": "timerTrigger",

"direction": "in",

"schedule": "0 \*/5 \* \* \* \*"

}

]

}

# Execute an Azure Function with triggers

[Click here | Determine the best trigger for your Azure function - Learn | Microsoft Docs](https://docs.microsoft.com/en-us/learn/modules/execute-azure-function-with-triggers/2-determine-best-trigger)

* + An Azure function can only have one trigger associated with it.
  + If it is required to invoke a function from multiple triggers, we need to create multiple azure functions with different triggers but same core function code.

# Chained Azure Functions: using input and output bindings

## Azure Cosmos DB

### Create a database account

[Azure portal](https://portal.azure.com/learn.docs.microsoft.com) -> HOME -> Create a resource -> Databases -> Azure Cosmos DB

-> Core (SQL) – Recommended -> Create -> input required values

{Subscription,

Resource Group,

Account Name = globally unique name (MyLearnAcc)

Location = region nearest to you}

-> Review + create -> Create -> Wait for deployment to complete -> Go to resource.

### Add a container

Azure Cosmos DB account -> Data Explorer -> New Container -> input required values

{Database id = “func-io-learn-db”

Container id = “Bookmarks”

Database Max RU/s = 4000

Partition key = “/id”}

-> OK

-> When complete, the Data Explorer displays func-io-learn-db in DATA under SQL API.

-> func-io-learn-db -> Bookmarks

### Add test data

func-io-learn-db -> Bookmarks -> Items -> New Item -> Replace JSON code

{

"id": "docs", // “Partition key given while creating container is necessary” //

"url": "https://docs.microsoft.com/azure"

}

-> Save

-> More properties appear; they are generated by the system to help manage the items in the container.

-> Create few more items

## Add an Azure Cosmos DB **input binding**

Function App Name (MyFuncApp) -> Functions -> Function Name (HttpTrigger2)

-> Integration -> Inputs -> Add input ->

{Binding Type = “Azure Cosmos DB”

Cosmos DB account connection = New link -> Install CosmosDB extension (If a message prompts)}

-> Azure Cosmos DB account = “MyLearnAcc” -> OK -> input required values

| **Setting** | **Value** | **Description** |
| --- | --- | --- |
| **Document parameter name** | bookmark | The name used to identify this binding in your code.  Parameter name from query string or request body |
| **Database name** | func-io-learn-db | The database to work with. This value is the database name we set. |
| **Collection Name** | Bookmarks | The collection from which we'll read data. This setting was defined. |
| **Document ID** | {id} | Add the Document ID that we defined when we created the *Bookmarks* Azure Cosmos DB container. |
| **Partition key** | {id} | Add the partition key that you defined when you created the *Bookmarks* Azure Cosmos DB collection.  The key entered here (specified in input binding format <key>) must match the one in the collection. |
| **SQL Query (optional)** | *Leave blank* | You are only retrieving one document at a time based on the ID. So, filtering with the Document ID setting is a better than using a SQL Query in this instance.  You could craft a SQL Query to return one entry (SELECT \* from b where b.ID = id). That query would indeed return a document, but it would return it in a document collection. Your code would have to manipulate a collection unnecessarily. Use the SQL Query approach when you want to get multiple documents. |

-> OK.

## Add an Azure Cosmos DB **output binding**

Function App Name (MyFuncApp) -> Functions -> Function Name (HttpTrigger2)

Integration -> Outputs -> Add output

{Binding Type = “Azure Cosmos DB”

Cosmos DB account connection = New link -> Install CosmosDB extension (If a message promps)}

-> Azure Cosmos DB account = “MyLearnAcc” -> OK -> input required values

| **Setting** | **Value** | **Description** |
| --- | --- | --- |
| **Document parameter name** | newbookmark | The name used to identify this binding in your code. This parameter is used to write a new bookmark entry. |
| **Database name** | func-io-learn-db | The database to work with. This value is the database name we set earlier in this lesson. |
| **Collection Name** | Bookmarks | The collection from which we'll read data. This is the name of the container that we defined earlier in the lesson. |
| **Partition key** | /id | Add the partition key that we defined when we created the Bookmarks Azure Cosmos DB container earlier. The key entered here (specified in input binding configuration <key>) must match the one in the container. |

-> OK.

## Add an Azure Queue Storage **output binding**

Function App Name (MyFuncApp) -> Functions -> Function Name (HttpTrigger2)

Integration -> Outputs -> Add output

{Binding Type = “Azure Queue Storage” -> Install Storage extension (If a message promps)}

-> Azure Cosmos DB account = “MyLearnAcc” -> OK -> input required values

| **Setting** | **Old value** | **New value** | **Description** |
| --- | --- | --- | --- |
| **Message parameter name** | outputQueueItem | newmessage | The binding property we'll use in code. |
| **Queue name** | outqueue | bookmarks-post-process | The name of the queue where we're placing bookmarks so that they can be processed further by another function. |

-> OK.

## Update the Azure function implementation

Function App Name (MyFuncApp) -> Functions -> Function Name (HttpTrigger2)

-> Developer -> ‘Code + Test’

-> Replace all the code in the index.js file with the code from the following snippet



-> Save.

Text

Description automatically generated

### What this code does:

1. Because this function changes our data, we expect the HTTP request to be a POST, and the bookmark data to be part of the request body.
2. Our Azure Cosmos DB input binding attempts to retrieve a document, or bookmark, by using the id that we receive.

If it finds an entry, the **‘req.bindings.bookmark’** object will be set.

1. The **if(bookmark)** condition checks to see whether an entry was found.
2. Adding to the database is as simple as setting the **‘context.bindings.newbookmark’** binding parameter to the value.

Here we set it to new bookmark entry, which we've created as a JSON string.

1. Posting a message to our queue is as simple as setting the ‘**context.bindings.newmessage’** parameter.

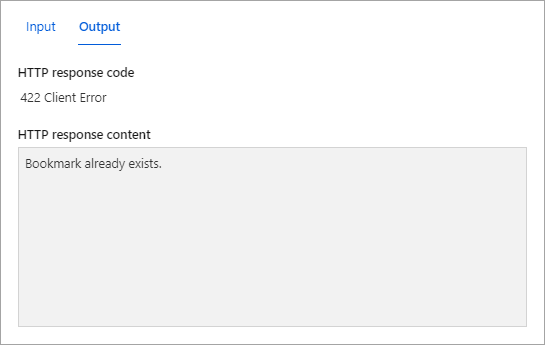
## Try it out

-> ‘Code + Test’ -> ‘Test/Run’ -> Below inputs -> Run.

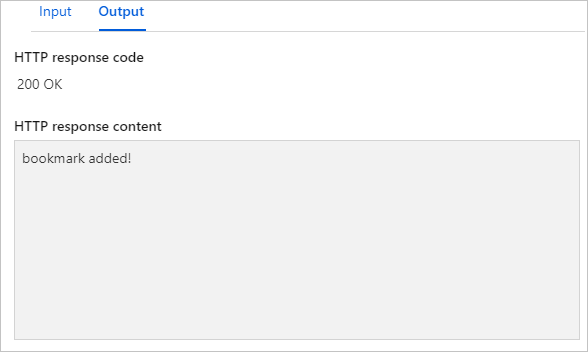
Graphical user interface, text, application, email

Description automatically generated

Output tab will display either



Or



### Verify that a message is written to the queue

[Azure portal](https://portal.azure.com/learn.docs.microsoft.com) -> HOME -> global search bar -> storage accounts

-> MyLearnAcc -> Data storage -> Queues -> bookmarks-post-process

-> ID and the Message text columns

Also look at the Cosmos database to verify that new entries have been added.

# Durable Functions: Create a long-running serverless workflow

## What is Durable Functions?

* Durable Functions is an extension of Azure Functions.
* Azure provides the infrastructure for maintaining state information.
* You can use Durable Functions to orchestrate a long-running workflow.
* You can implement common patterns such as fan-out/fan-in.
* Azure checkpoints the progress of a function automatically when the function awaits. Azure may choose to dehydrate the function and save its state while the function waits, to preserve resources and reduce costs. When the function starts running again, Azure will rehydrate it and restore its state.

## Function types

### **Client functions**

These are the entry point for creating an instance of a Durable Functions orchestration. They can run in response to an event from many sources, such as a new HTTP request arriving, a message being posted to a message queue, an event arriving in an event stream. You can write them in any of the supported languages.

### Orchestrator functions

These describe how actions are executed, and the order in which they are run. You write the orchestration logic in code (C# or JavaScript).

### Activity functions

These are the basic units of work in a durable function orchestration. An activity function contains the actual work performed by the tasks being orchestrated.

### Entity functions

Entity functions define operations for reading and updating small pieces of state. We often refer to these stateful entities as durable entities. Like orchestrator functions, entity functions are functions with a special trigger type,***entity trigger***. They can also be invoked from client functions or from orchestrator functions. Unlike orchestrator functions, entity functions do not have any specific code constraints. Entity functions also manage state explicitly rather than implicitly representing state via control flow.

#### Note:

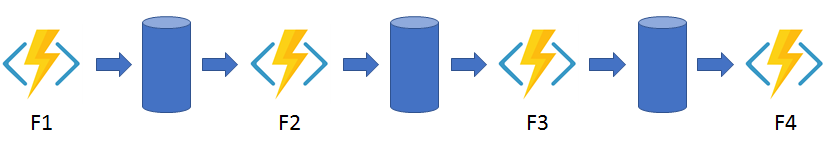
* Entities are accessed via a unique identifier, the *entity ID*. An entity ID is simply a pair of strings that uniquely identifies an entity instance.
* Operations on entities require that you specify the **Entity ID** of the target entity, and the **Operation name**, which is a string that specifies the operation to perform.

## Application patterns

You can use Durable Functions to implement many common workflow patterns. These patterns include:

### Function chaining

In this pattern, the workflow executes a sequence of functions in a specified order. The output of one function is applied to the input of the next function in the sequence. The output of the final function is used to generate a result.



### Fan out/fan in

This pattern runs multiple functions in parallel and waits for all the functions to finish. You can aggregate the results of the parallel executions or use them to compute a final result.



### Async HTTP APIs

This pattern addresses the problem of coordinating state of long-running operations with external clients. An HTTP call can trigger the long-running action. Then it can redirect the client to a status endpoint. The client can learn when the operation is finished by polling this endpoint.

Icon

Description automatically generated with medium confidence

### Monitor

This pattern implements a recurring process in a workflow, possibly looking for a change in state. For example, you could use this pattern to poll until specific conditions are met.

### Human interaction

This pattern combines automated processes that also involve some human interaction. A manual process within an automated process is tricky because people aren't as highly available and as responsive as most computers. Human interaction can be incorporated using timeouts and compensation logic that runs if the human fails to interact correctly within a specified response time. An approval process is an example of a process that involves human interaction.



## Comparison with Logic Apps

* You can use either technology to create serverless complex orchestrations.
* Azure Durable Functions is intended as a powerful serverless compute option to run custom logic. Azure Logic Apps is better suited for integrating Azure services and components.
* With Azure Durable Functions, you develop orchestrations by writing code and using the Durable Functions extension.
* With Logic Apps, you create orchestrations by using the design surface or editing configuration files.

| **Task** | **Azure Durable Functions** | **Azure Logic Apps** |
| --- | --- | --- |
| Development | Code-first (imperative) | Design-first (declarative) |
| Connectivity | About a dozen built-in binding types. You can write code for custom bindings. | Large collection of connectors. Enterprise Integration Pack for B2B. You can also build custom connectors. |
| Actions | Each activity is an Azure Function. You write the code for activity functions. | Large collection of ready-made actions. You integrate custom logic through custom connectors. |
| Monitoring | Azure Application Insights | Azure portal, Azure Monitor logs |
| Management | REST API, PowerShell, Visual Studio | Azure portal, REST API, PowerShell, Visual Studio, [Visual Studio Code extension](https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurelogicapps#:%7E:text=%20Run%20the%20logic%20app%20locally%20%201%2cedit%20the%20value%20of%20AzureWebJobsStorage%20in...%20More%20) |

## Design a workflow based on Durable Functions

The workflow steps are as follows:

A project design is submitted.

An approval task is allocated to a manager so they can review the project design proposal.

The project design proposal is rejected or approved.

An escalation task is allocated if the approval task isn't completed within a pre-defined time limit.

The following table shows how the workflow steps can be mapped to the function types we use in a Durable Functions workflow.

| **Workflow function** | **Durable Function Type** |
| --- | --- |
| Submitting a project design proposal for approval | *Client* Function |
| Assign an Approval task to relevant member of staff | *Orchestration* Function |
| Approval task | *Activity* Function |
| Escalation task | *Activity* Function |

## Create a workflow using Durable Functions

Create an approval workflow in the Azure portal using Durable Functions.

### Create a Function App

As directed in previous module – Create a function app in the Azure portal

### Install the durable-functions npm package

Function App -> under Development Tools -> App Service Editor (preview) -> Go

-> In the left menu pane, highlight the ‘WWWROOT’ folder

-> In the left toolbar menu, select the Open Console icon -> execute below commands

-> Create a new package.json file ***touch package.json***

-> Open ***open package.json***

-> Edit – replace by this Json { "name": "example",//Any name

"version": "1.0.0" }

-> Save ***Ctrl + S***

-> Close ***Ctrl + Q***.

-> Switch back to the Azure portal

-> under Development Tools -> Console

-> verify path “*C:\home\site\wwwroot*”

-> Run the following command: ***npm install durable-functions***

-> In the left menu pane, scroll up -> Overview -> Restart -> Yes.

### Add moment npm package to your function app

Function App -> under Development Tools -> Console

-> verify path “C:\home\site\wwwroot”

-> Run the following command to install the typescript library **npm install typescript**

-> Run the following command to install the moment library **npm install moment**

### Create the client function for submitting a design proposal

Function App -> In the left menu pane -> Functions -> Create -> Below inputs

{Template = Durable Functions HTTP starter

name = HttpStart

Authorization level field = Function}

-> Create

### Create the orchestrator function

Function App -> In the left menu pane -> Functions -> Create -> Below inputs

{Template = Durable Functions orchestrator

name = OrchFunction}

-> Create



* If the Approval function doesn't respond within 20 secs, the Escalation function is called.
* The code waits for an external input to call Approval function.
* This way we can control when the response comes back for testing purposes.

### Create the Approval activity function

Function App -> In the left menu pane -> Functions -> Create -> Below inputs

{Template = Durable Functions activity

name = Approval}

-> Create



* The expression context.bindings.name will either be Accepted or Rejected.
* Depending on this parameter passed from the orchestrator function, activity function will execute corresponding action.

### Add an escalation activity to your function app

Function App -> In the left menu pane -> Functions -> Create -> Below inputs

{Template = Durable Functions activity

name = Escalation}

-> Create



* This function would contain the logic to remind the recipient and / or reassign the task.

## Verify that the Durable Functions workflow starts

Function App -> In the left menu pane -> Overview -> Restart -> Yes

-> In the left menu pane -> Functions -> HttpStart

-> On the top menu bar -> Get Function URL -> copy the URL

Your URL should resemble this:

[https://example.azurewebsites.net/api/orchestrators/{functionName}?code=AbCdEfGhIjKlMnOpQrStUvWxYz==](https://example.azurewebsites.net/api/orchestrators/%7bfunctionName%7d?code=AbCdEfGhIjKlMnOpQrStUvWxYz==)

In the URL, replace the {functionName} placeholder with Name of your orchestration function (OrchFunction in this case).

The response message contains a set of URI endpoints that you can use to monitor and manage the execution, which should resemble the following example.

{

"id": "f0e1d2c3b4a5968778695a4b3c2d1e0f",

"statusQueryGetUri": "https://example.azurewebsites.net/...",

"sendEventPostUri": "https://example.azurewebsites.net/...",

"terminatePostUri": "https://example.azurewebsites.net/...",

"rewindPostUri": "https://example.azurewebsites.net/...",

"purgeHistoryDeleteUri": "https://example.azurewebsites.net/..."

}

**StatusQueryGetUri** -> Execute in browser

-> **response** message as below:

While it is waiting for the timer to countdown to 20 seconds

{ status = **Running** , **output = null }**

After **waiting 20 seconds**

{

status = **Completed**,

**output** = [ "ESCALATION : You have not approved the project design proposal - reassigning to your Manager! Head of department!" ]

}

# Develop, test, and publish Azure Functions using Azure Functions Core Tools

## What are the Azure Functions Core Tools?

The Azure Functions Core Tools are command-line utilities that let you develop and run functions locally, and then publish them to Azure.

### The Core Tools feature different kinds of functions-related capabilities, but their primary purpose is to:

* + - 1. Generate the files and folders you need to develop functions on your local computer.
      2. Run your functions locally so you can test and debug them.
      3. Publish your functions to Azure.

## Create and run Azure Functions locally by using the Core Tools

### Prerequisites:

Cloud Shell environment, and Core Tools are already installed,

The Azure CLI and a code editor are already installed.

### Function apps and functions projects

A functions project on your computer is equivalent to a function app in Azure, and can contain multiple functions that use the same language runtime.

### Create a new functions project

CLI

-> func init -> 2. Node -> 1. JavaScript

### Create a new function

CLI

-> func new -> 8. Http Trigger -> name = simple-interest

### Implement the simple-interest function

Open the Cloud Shell editor by

CLI -> code .



### Run the function locally

CLI

-> func start ->

Near the end of the output, you'll see a message that lists Functions:

simple-interest: is available as a GET or POST HTTP request <http://localhost:7071/api/simple-interest>

-> curl "http://localhost:7071/api/simple-interest?principal=5000&rate=.035&term=36" -w "\n"

Stop the background Functions host by

-> pkill func

view the output log

-> code ~/output.txt

Close the editor

Ctrl + Q

## Publish a function to Azure using Core Tools

### Create a function app in Azure

CLI ->

* 1. Assign RESOURCEGROUP, STORAGEACCT & FUNCTIONAPP names to shell variables.

RESOURCEGROUP="[sandbox resource group]"

STORAGEACCT=learnstorage$(openssl rand -hex 5)

FUNCTIONAPP=learnfunctions$(openssl rand -hex 5)

*$(openssl rand -hex 5) generates a random five-character string, to ensure that the names meet the requirement of being globally unique.*

* 1. Create an Azure Storage Acc & a Function App

az storage account create \

--resource-group "$RESOURCEGROUP" \

--name "$STORAGEACCT" \

--kind StorageV2 \

--location centralus

az functionapp create \

--resource-group "$RESOURCEGROUP" \

--name "$FUNCTIONAPP" \

--storage-account "$STORAGEACCT" \

--runtime node \

--consumption-plan-location centralus \

--functions-version 3

### Publish to Azure

CLI

-> cd ~/loan-wizard -> func azure functionapp publish "$FUNCTIONAPP" –force

-> Select the invoke URL from the previous command's output to open it in a new browser tab.

-> Add &principal=5000&rate=.035&term=36 to the end of the URL.

# Develop, test, and deploy an Azure Function with Visual Studio

## Create an Azure Function App

Visual Studio Installer -> Visual Studio 2019 start page -> select Modify

-> ASP.NET and web development & Azure development -> Install while downloading (If not already available) -> Close.

Launch Visual Studio 2019 -> Create a new project -> Azure Functions -> Next

| **Field** | **Value** |
| --- | --- |
| **Project name** | *WatchPortalFunction* |
| **Location** | Browse to or create a folder on your local computer to store the project. |
| **Solution name** | Accept *WatchPortalFunction* (default). |
| **Place solution and project in the same directory** | Unchecked (NOT selected). This ensures the correct folder structure for this module. |

-> Create

| **Field** | **Value** |
| --- | --- |
| **Dotnet version** | *.NET Core 3 (LTS)* or *.NET Core 2* |
| **Function trigger** | *Http trigger* |
| **Storage account** | *Storage emulator* |
| **Authorization level** | *Anonymous* |

-> If a Visual Studio notification shows that updates are ready, select Refresh

-> Create.

## Create the WatchInfo Azure Function

Solution Explorer -> Right click the project name -> Add -> New Item -> Azure Function

-> Name: WatchInfo.cs -> Add

-> Http trigger -> Authorization level -> Anonymous -> Add

## Test the Azure Function locally

Run and debug solution in VS normally.

## Publish a simple Azure Function

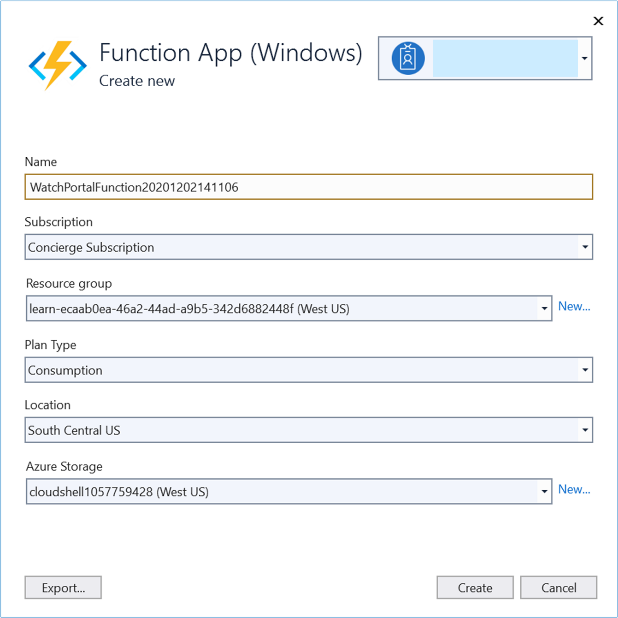
### Create a function app in the Azure portal

### Deploy the WatchInfo function to the Azure Function App

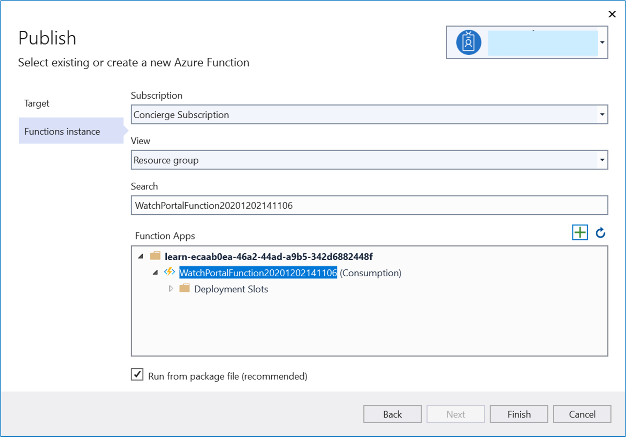
Solution Explorer -> Right click the project name -> Publish -> Azure -> Next

{ Subscription, Resource Group }

-> (+)



-> Create



-> Finish -> Publish -> Output window

If everything works, the final two messages should be Build succeeded and Publish succeeded.

### Test your Azure function

# Webhooks : Monitor GitHub events using Azure Functions

## What is a webhook?

### Webhooks are user-defined HTTP callbacks. They're triggered by some event, such as pushing code to a repo or updating a wiki page. When the event occurs, the source site makes an HTTP request to the URL configured for the webhook.

### With Azure Functions, we can define logic in a function that can be run when a webhook message is received.

## Create an Azure function triggered by webhook

### [Create an Http Triggered Function App](#_Execute_an_Azure)

### Set up a webhook for a GitHub repository

In the GitHub portal -> Your repository -> Settings -> Webhooks -> Add webhook

“Payload URL” = your function app url to receive webhook post requests

“Content type” = “application/json”

-> Let me select individual events

-> select the **Wiki** checkbox. Make sure no other checkboxes are selected

-> select Active checkbox

-> Add webhook

### Parse information from the Gollum event

Update your function app code as below as per the event payload to parse the information specific to the event.



### Trigger your Azure Function with a Gollum event

Your GitHub account -> Your repository -> Settings -> Webhooks -> Edit (against your webhook)

-> Manage webhook Pane appears -> Recent Deliveries

-> Select the latest (top) delivery entry by selecting its ellipsis button (...)

-> Redeliver -> Yes, redeliver this payload

-> Select the (redelivery) latest (top) delivery entry by selecting its ellipsis button (...)

-> Response tab -> Page is Home, Action is edited, Event Type is gollum

## Secure Webhook payloads with a secret

### Get a key for your Azure Function



**Functions -> HtttpTrigger1 -> Developer -> Function Keys**

**-> Under the Value column, select the Hidden value. Click to show value link**

**-> Copy to clipboard**

**-> Replace the <default key> in index.js with the default key that you just copied to the clipboard**

**This code computes the hash of the key, using the same mechanism as GitHub.**

### Update the webhook secret

Your repository -> Settings -> Webhooks

-> Edit (against your webhook)

-> Secret = default key value from your function app

-> Update webhook

### Test the webhook and the Azure Function

[Trigger webhook](#_Trigger_your_Azure)

### Test an invalid signature

Settings -> **Secret -> Change Secret -> Update webhook.**

**->** [Trigger webhook](#_Trigger_your_Azure)

# SignalR Service : Enable automatic updates in a web application using Azure Functions

Your objective is to update the app to implement automatic updates of the stock price information, but ensure communication between the client and server happens only when data changes on the server.

## Analyze the limitations of a polling-based web app

1. [Download the sample code](https://github.com/MicrosoftDocs/mslearn-advocates.azure-functions-and-signalr)
2. [Create a Storage account](#_Create_a_storage)
3. [Create an Azure Cosmos DB account](#_Azure_Cosmos_DB)
4. Update storage & CosmosDB account connection strings and CosmosDB account key in local.settings.json file.
5. Run the application
   1. npm install
   2. Press F5 – to start debugging the function app
   3. npm start – to start the web app
   4. <http://localhost:8080> – to view the web page in browser
   5. npm run update-data – to update data in CosmosDB
6. Observe updated data reflects in the browser page only after few seconds.
7. Stop the running processes
   1. kill process (trash can icon) – to stop the web server
   2. Stop or press Shift+F5 – to stop the functions app

SignalR and persistent connections

In contrast to polling, a more favorable design features persistent connections between the client and server.

**Establishing a persistent connection allows the server to push data to the client at will.**

**The on-demand nature of the connection reduces network traffic and load on the server.**

SignalR handles connection management automatically, and lets you **broadcast messages to all connected clients simultaneously**, like a chat room. You can also send messages **to specific clients**.

SignalR is an abstraction for a series of technologies that allows your app to enjoy two-way communication between the client and server.

A key benefit of the abstraction provided by SignalR is the way it **supports "transport" fallbacks**.

A transport is method of communicating between the client and server. SignalR connections begin with a standard HTTP request. As the server evaluates the connection, the most appropriate communication method (transport) is selected. **Transports are chosen depending on the APIs available on the client.**

* For clients that support **HTML 5, the WebSockets API transport** is used by default.
* If the client **doesn't support WebSockets, then SignalR falls back to Server Sent Events** (also known as EventSource).
* **For older clients, Ajax long polling** or Forever Frame (IE only) is used to mimic a two-way connection.

## Enable automatic updates in a web application using SignalR Service

1. **Create a SignalR account** on Azure portal.
2. **In local.settings.json, update** the variable AzureSignalRConnectionString.
3. **Create an Http Trigger Azure Function** named “negotiate” to return the SignalR connection info.

to broadcast changes to connected clients using SignalR Service.

* 1. negotiate/function.json -> Add to bindings array

{

"type": "signalRConnectionInfo",

"name": "connectionInfo",

"hubName": "stocks",

"direction": "in",

"connectionStringSetting": "AzureSignalRConnectionString"

}

* 1. negotiate/index.js -> Replace existing code with following code

module.exports = async function (context, req, connectionInfo) {

context.res.body = connectionInfo;

};

1. **Create an Azure Cosmos DB Trigger Azure function** named “stocksChanged” to Detect and broadcast database changes.
   1. stocksChanged/function.json -> existing trigger binding definition

-> Append the property "feedPollDelay": 500

* 1. Next, append the following SignalR output binding definition to the bindings array.

{

"type": "signalR",

"name": "signalRMessages",

"hubName": "stocks",

"direction": "out",

"connectionString": "AzureSignalRConnectionString"

}

* 1. stocksChanged/index.js ->

module.exports = async function (context, documents) {

const updates = documents.map(stock => ({

target: 'updated',

arguments: [stock]

}));

context.bindings.signalRMessages = updates;

context.done();

}

1. **Update the web application**

public/index.html

public/index.html.js

[Find the code here.](https://github.com/AjinkyaApte88/General/tree/main/mslearn-azure-functions-and-signalr/start)

### What this code does?

**As the client receives messages from the server, it listens for messages via the on('updated',...) syntax. Once an update is received, the following actions take place:**

* **The changed stock is located in the array.**
* **The previous version is removed.**
* **The updated version is inserted at the same index position in the array.**

**Manipulating the array this way allows Vue to detect changes in the data and trigger animation effects to notify users of changes.**

1. Run the application
   1. npm install
   2. Press F5 – to start debugging the function app
   3. npm start – to start the web app
   4. <http://localhost:8080> – to view the web page in browser
   5. npm run update-data – to update data in CosmosDB
2. Observe updated data reflects in the browser page just over half second later.
3. Stop the running processes
   1. kill process (trash can icon) – to stop the web server
   2. Stop or press Shift+F5 – to stop the functions app

## Use a storage account to host a static website

1. Deploy the function app
2. public/index.html.js -> replace <FUNCTION\_APP\_ENDPOINT> with the function's endpoint.
3. F1 -> Azure Functions: Upload local settings
4. Configure static websites in Azure Storage

| **Name** | **Value** |
| --- | --- |
| Storage account | Select the account you created earlier. |
| Default file | Select **index.html** as the index document name for the account. |
| Error document | Enter **index.html** for the default 404 error document path. |

1. Deploy the web application to Azure Storage
2. Determine the primary endpoint address of the static website

Azure portal -> your storage account -> your static website primary endpoint

1. Set up CORS in the function app

Azure portal -> your function app -> Left menu -> API -> CORS

-> Select Enable Access-Control-Allow-Credentials

-> add entry -> static website primary endpoint (remove the trailing /)

## Run the deployed application

1. Arrange Visual Studio Code on one side of the screen and the web browser running the static site on the other.
2. Refresh the browser. It may take a moment for stocks to appear as the serverless functions are running for the first time.
3. In Visual Studio integrated terminal -> npm run update-data
4. Observe as the UI is automatically updated

# Build serverless apps with Go

## Custom handlers

Azure Functions features a variety of language runtimes. If your language of choice is not provided by default, you can use a custom handler.

At its core, a custom handler is a web server. The web server receives events from the Functions host. You then have an opportunity to write code in your preferred language to respond to the events.

With custom handlers, you can use any language that supports HTTP primitives. That's nearly any language.

Diagram

Description automatically generated

## Use a custom handler to build an app

### Scaffold the app

Using the Azure Functions extension in Visual Studio Code.

View -> Command Palette -> “Azure Functions: Create New Project” -> Select a folder

-> From “Select a language” list, select ‘Custom Handler’.

-> select HttpTrigger -> enter name as ‘hello’ -> select auth level anonymous

### Create the app

Create a file named server.go at the project root.



### Run the app

From a terminal, run  ‘**go build server.go**’ from project root.

Open the ‘host.json’ file and find the “defaultExecutablePath” element inside the “customHandler” element.

Specify “./server” on macOS and Linux, or “.\\server.exe" on a Windows OS.

Under the customHandler element, add the “enableForwardingHttpRequest” element and give it the value true.

From a terminal, run **func start** in the project root. Doing so starts your Functions app.

At the end of the output, you'll see an output similar to:

Functions:

hello: [GET,POST] {URL}

In a browser, go to output URL.

You should see the output "hello world."

## Check your knowledge

**How can you make Azure Functions find your compiled app?**

- Set defaultExecutablePath in the host.json file.

**Can any language or runtime use custom handlers?**

- Can any language or runtime use custom handlers?