# Identify the technology options

Modern businesses run on multiple applications and services. How well your business runs can often be impacted by how efficiently you can distribute the right data to the right task. Automating this flow of data can streamline your business even further. Choosing the right technology for these critical data integrations and process automation is also an important consideration.

The first question to ask is whether you prefer to design the workflow in a GUI designer tool or by writing code.

The following list has some valid reasons for using a design-first tool:

* People who design the workflow have no coding experience.
* Later designers and users can consult the graphical design to clearly understand how the workflow proceeds.

Alternatively, you can choose to use a code-first tool because:

* People who design the workflow are developers and prefer to work entirely in code.
* You want the details of a workflow to be hidden from non-coders.

Diagram

Description automatically generated

## Analyze the decision criteria

### Design First

* + In **Logic Apps**, there is a GUI designer on which you draw out the workflow. It is intuitive and easy to use but you also can delve under the hood and edit the source code for a workflow. This tool is designed for people with development skills.
  + In **Microsoft Power Automate**, extra help and templates are provided for common types of workflows. There is no way to edit the source code that the tool creates. This tool is designed for users who have a good understanding of the business process but no coding skills.

### Code First

* + Because of the extra features that are included with **Azure Functions**, including wider ranges of trigger events and supported languages, the ability to develop test code in the browser, and the pay-per-use price model, consider Azure Functions to be your default choice.
  + **Web jobs** only supports C# on Microsoft Windows.

There are two situations in which web Jobs might be a better choice:

* + - You have an existing Azure App Service application, and you want to model the workflow within the application. This requirement means that the workflow can also be managed as part of the application, for example in an Azure DevOps environment.
    - You have specific customizations that you want to make to the JobHost that are not supported by Azure Functions. For example, in a Web Job, you can create a custom retry policy for calls to external systems. This kind of policy can't be configured in an Azure Function.

### Azure Functions or Azure Apps Service Web Jobs?

* Cost: With Web Jobs, you pay for the entire VM or App Service Plan that hosts the job. Azure Function can run on a consumption plan, so you only pay when the function runs. Since this process only kicks off when a bike is returned, we might stand to save by selecting Azure Functions.
* Integrations: You want to integrate the maintenance workflow with the Logic App that you build for the bike booking and rental process in the previous unit. Although it is possible to call a Web Job from a Logic App, the integration between Logic Apps and Functions is closer. For example, you can more easily control your call to a Function from the Logic Apps designer.

## Mixing technologies

Remember that there is no need for you to use the same technology for different workflows:

If your requirements differ, you are likely to reach a different answer at the end of your decision-making process.

Furthermore, you can also call one workflow from another. For example, a workflow implemented in Microsoft Power Automate can easily call another that is built as an Azure Function.

One reason to mix the technologies used in your business processes would be to give users control over a small section of a complete workflow. Do this by implementing that section in Microsoft Power Automate, then call that flow from a Logic App, Web Job, or Function.

# 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.

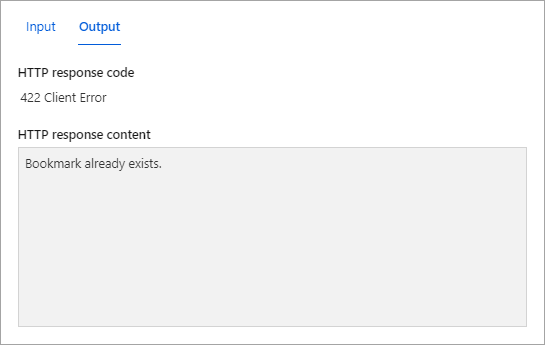
## Try it out

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

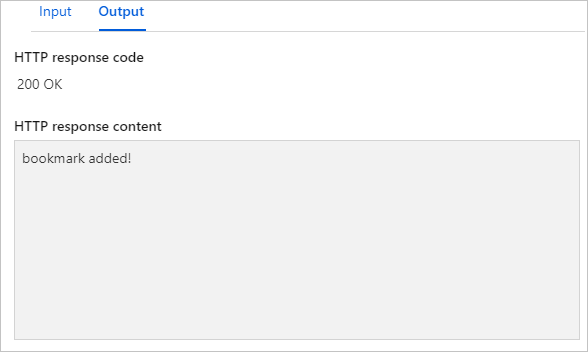
Graphical user interface, text, application, email

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

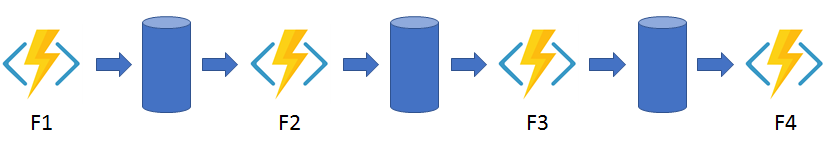
These are the basic units of work in a durable function orchestration. An activity function contains the actual work performed by the tasks being orchestrated.

## 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

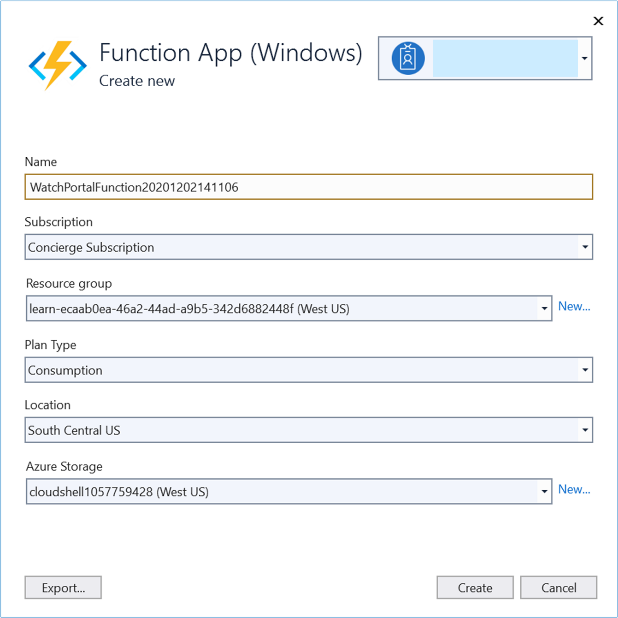
### Create a function app in the Azure portal

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

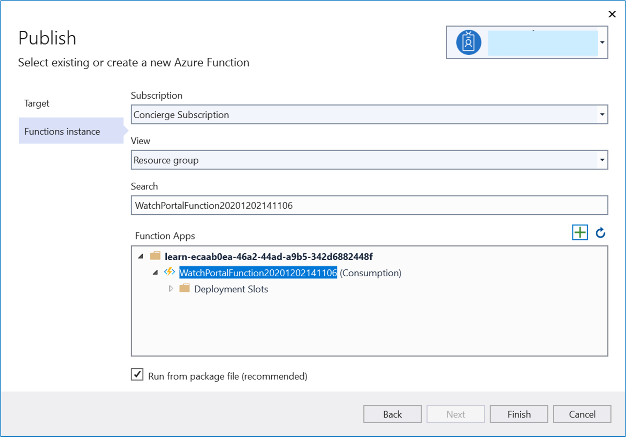
Solution Explorer -> Right click the project name -> Publish -> Azure -> Next

{ 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

# Monitor GitHub events by using a webhook with 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 a 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)

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

# Expose multiple Azure Function apps as a consistent API by using Azure API Management

## Azure API Management

Azure API Management is a fully managed cloud service that you can use to publish, secure, transform, maintain, and monitor APIs. It helps organizations publish APIs to external, partner, and internal developers to unlock the potential of their data and services. API Management handles all the tasks involved in mediating API calls, including request authentication and authorization, rate limit and quota enforcement, request and response transformation, logging and tracing, and API version management. API Management enables you to create and manage modern API gateways for existing backend services no matter where they're hosted.

### Microservices architecture challenges

Client apps are coupled to microservices. If you want to change the location or definition of the microservice, you may have to reconfigure or update the client app.

Each microservice may be presented under different domain names or IP addresses. This presentation can give an impression of inconsistency to users and can negatively affect your branding.

It can be difficult to enforce consistent API rules and standards across all microservices. For example, one team may prefer to respond with XML and another may prefer JSON.

You're reliant on individual teams to implement security in their microservice correctly. It's difficult to impose these requirements centrally.

### API Management also includes helpful tools

You can test each microservice and its operations to ensure that they behave in accordance with your requirements. You can also monitor the behavior and performance of deployed services.

Azure API Management supports importing Azure Function Apps as new APIs or appending them to existing APIs. The process automatically generates a host key in the Azure Function App, which is then assigned to a named value in Azure API Management.

## Create a new API in API Management from a function app

### Create functions

[Create an Azure Function App - ProductFunction](#_Create_a_function)

[Create ProductDetails function in your ProductFunction app](#_Create_a_function_1)

[Test ProductDetails function](#_Test_your_Azure)

### Expose function app as an API using Azure API Management

Azure Portal -> All resources -> ProductFunction function app

-> API -> API Management -> Create new

-> Install API Management gateway pane appears

“Subscription” = Concierge Subscription,

“Resource group” = [sandbox resource group name]

“Region” = select any that supports the Consumption Plan

“Administrator email” = Enter an email address.

“Pricing tier” = Consumption (99.95% SLA)

-> Next: Monitoring and clear the Application Insights option

-> Review + Create -> Create -> Link API

-> Import Azure Functions API Management service pane appears

-> Select “ProductDetails” function and Select to continue

-> “API URL suffix” = “products”

-> Create.

### Test the OnlineStore products endpoint

API Management pane of your function app -> Test -> GET ProductDetails

-> Under Query parameters, select Add parameter

-> Enter name in the NAME field and value and in the VALUE field

-> Send

-> The **HTTP response** section of the console has two tabs, Message and Trace. The Message tab is populated with the HTTP response. The product details appear in JSON format at the end of the response.

-> Scroll up to the HTTP request section and notice the format of the request. The request was sent to a destination in the azure-api.net domain. This location is different from the azurewebsites.net domain where the function app is hosted.

## Add another Azure Functions app to an existing API

### Create another function

[Create OrderDetails function in your ProductFunction app](#_Create_a_function_1)

[Test the OrderDetails function](#_Test_your_Azure)

### Add a function to an existing API

Azure Portal -> All resources -> OnlineStore API Management service

-> APIs -> APIs -> Create from Azure resource -> Function App

-> Browse -> Import Azure Functions pane appears

-> Select -> Select Azure Function App pane appears

-> Select OrderFunction\*\*\*\*\*\*\* App -> Select -> Select OrderDetails function -> Select

-> “API URL suffix” = “orders” -> Create

### Test the OnlineStore orders endpoint in the portal

[Test orders endpoint](#_Test_the_OnlineStore_1)

Notice that both the functions can now be called through endpoints within the **azure-api.net** domain (as defined by the GATEWAY\_URL), which is the domain used by Azure API Management.

In other Learn modules**, you can learn how to**

* + - **apply policies,**
    - **security settings,**
    - **external caches,**
    - **and other features to the functions in an API Management Gateway.**

**APIM gateway provides you with a central control point, where you can manage multiple microservices without altering their code.**

# 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?

# Choose a messaging model in Azure to loosely connect your services

Many applications consist of programs that run on several different computers or devices. In such distributed applications, messages must be sent between the components across networks and long distances. Even on the same server or in the same data center, loosely coupled architectures require mechanisms for components to communicate. Reliable messaging is often a critical problem.

You plan to solve these issues by using one or more of the following technologies:

1. Azure Storage queues,
2. Azure Event Hubs,
3. Azure Event Grid,
4. Azure Service Bus.

## Choose whether to use messages or events

### What is a message?

* contains raw data, **produced by one** component, that will be **consumed by another** component.
* contains the **data itself**, not just a reference to that data.
* The sending component **expects** the destination component to **process** the message content **in a certain way**

### What is an event?

* Events are **lighter weight** notification that indicates that something happened
* Events are most **often used for broadcast communications**
* event may be sent to multiple receivers, or to none at all
* Events are often intended to "**fan out**," or have **a large number of** **subscribers** for each publisher
* The publisher of the event has **no expectation** about the action a receiving component takes
* Some events are discrete units and unrelated to other events
* Some events are part of a related and ordered series

### How to choose messages or events

For each communication, consider the following question:

#### Does the sending component expect the communication to be processed in a particular way by the destination component?

## Azure message-based delivery with queues

Suppose you are planning the architecture for your music-sharing application. You want to ensure that music files are uploaded to the web API reliably from the mobile app. You then want to deliver the details about new songs directly to the app when an artist adds new music to their collection. This scenario is a perfect use of a message-based system and Azure offers two solutions to this problem:

1. Azure Queue Storage, 2. Azure Service Bus Queues, 3. Azure Service Bus Topics

### What is Azure Queue Storage?

Queue storage is a service that uses Azure Storage to store large numbers of messages that can be securely accessed from anywhere in the world using a simple REST-based interface. Queues can contain millions of messages, limited only by the capacity of the storage account that owns it.

### What is Azure Service Bus Queues?

Service Bus is a message broker system intended for enterprise applications. These apps often utilize multiple communication protocols, have different data contracts, higher security requirements, and can include both cloud and on-premises services. Service Bus is built on top of a dedicated messaging infrastructure designed for exactly these scenarios.

### What are Azure Service Bus Topics?

Azure Service Bus topics are like queues, but can have multiple subscribers. When a message is sent to a topic instead of a queue, multiple components can be triggered to do their work.

Internally, topics use queues. When you post to a topic, the message is copied and dropped into the queue for each subscription. The queue means that the message copy will stay around to be processed by each subscription branch even if the component processing that subscription is too busy to keep up.

## Benefits of queues

### Increased reliability

Queues increase the reliability of the message exchange because, at times of high demand, messages can wait until a destination component is ready to process them.

### Message delivery guarantees

Queuing systems usually guarantee delivery of each message in the queue to a destination component. However, these guarantees can take different approaches:

1. **At-Least-Once Delivery**
2. **At-Most-Once Delivery**
3. **First-In-First-Out (FIFO)**
4. **Transactional support**

## Choose a service

### Use Service Bus topics if you:

* Need multiple receivers to handle each message

### Use Service Bus queues if you:

* Need an At-Most-Once delivery guarantee.
* Need a FIFO guarantee.
* Need to group messages into transactions.
* Want to receive messages without polling the queue.
* Need to provide a role-based access model to the queues.
* Need to handle messages larger than 64 KB but less than 100 MB. The maximum message size supported by the standard tier is 256 KB and the premium tier is 100 MB.
* Know Queue size will not grow larger than 1 TB. The maximum queue size for the standard tier is 80 GB and for the premium tier, it's 1 TB.
* Want to publish and consume batches of messages.

### Use Queue storage if you:

* Need an audit trail of all messages that pass through the queue.
* Expect the queue to exceed 1 TB in size.
* Want to track progress for processing a message inside of the queue.

**Use Storage queues when you want a simple and easy-to-code queue system.**

**For more advanced needs, use Service Bus queues.**

**If you have multiple destinations for a single message, but need queue-like behavior, use Service Bus topics.**

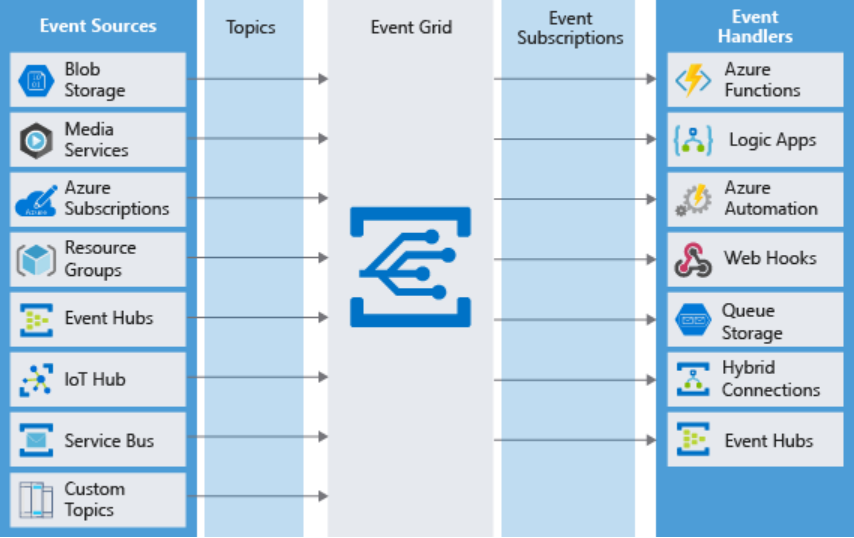
## Choose Azure Event Grid

### What is Azure Event Grid?

* Event Grid distributes events from different sources, such as Azure Blob storage accounts or Azure Media Services, to different handlers, such as Azure Functions or Webhooks.
* Event Grid was created to make it easier to build event-based and serverless applications on Azure.
* Event Grid supports most Azure services as a publisher or subscriber and can be used with third-party services.
* It provides a dynamically scalable, low-cost, messaging system that allows publishers to notify subscribers about a status change.

### Azure Event Grid Components:

* Events: What happened.
* Event sources: Where the event took place.
* Topics: The endpoint where publishers send events.
* Event subscriptions: The endpoint or built-in mechanism to route events, sometimes to multiple handlers. Subscriptions are also used by handlers to filter incoming events intelligently.
* Event handlers: The app or service reacting to the event.



### What is an event?

Events are the data messages passing through Event Grid that describe what has taken place.

Each event is self-contained, can be up to 64 KB, and contains several pieces of information based on a schema defined by Event Grid:

[ {

// The full resource path to the event source. Event Grid provides this value.

"topic": string,

// Publisher-defined path to the event subject.

"subject": string,

// The unique identifier for event.

"id": string,

// One of the registered event types for this event source. This is a value you can create filters // against, e.g. CustomerCreated, BlobDeleted, HttpRequestReceived, etc.

"eventType": string,

// The time the event was generated based on the provider's UTC time.

"eventTime": string,

// Specific information that is relevant to the type of event.

// However, the actual object that was changed is not part of the event data. Instead, a URL or / // identifier is often passed to reference the changed object.

"data":{

object-unique-to-each-publisher

},

"dataVersion": string,

"metadataVersion": string

} ]

### When should you use Event Grid?

Use Event Grid when you need these features:

* Simplicity: It is straightforward to connect sources to subscribers in Event Grid.
* Advanced filtering: Subscriptions have close control over the events they receive from a topic.
* Fan-out: You can subscribe to an unlimited number of endpoints to the same events and topics.
* Reliability: Event Grid retries event delivery for up to 24 hours for each subscription.
* Pay-per-event: Pay only for the number of events that you transmit.

**Event Grid is a simple but versatile event distribution system. Use it to deliver discrete events to subscribers, which will receive those events reliably and quickly.**

## Choose Azure Event Hubs

There are certain applications that produce a massive number of events from almost as many sources. We often hear the term "Big Data" applied to these situations, and they require unique infrastructure to handle them.

### What are Azure Event Hubs?

Event Hubs is an intermediary for the publish-subscribe communication pattern.

Unlike Event Grid, however, it is optimized for extremely high throughput, a large number of publishers, security, and resiliency.

Whereas Event Grid fits perfectly into the publish-subscribe pattern in that it simply manages subscriptions and routes communications to those subscribers, **Event Hubs performs quite a few** **additional services**.

#### Partitions:

As Event Hubs receives communications, it divides them into partitions.

Partitions are buffers into which the communications are saved.

Because of the event buffers, events are not completely ephemeral, and an event isn't missed just because a subscriber is busy or even offline. The subscriber can always use the buffer to "catch up".

#### Capture:

Event Hubs can send all your events immediately to Azure Data Lake or Azure Blob storage for inexpensive, permanent persistence.

#### Authentication:

All publishers are authenticated and issued a token. This means Event Hubs can accept events from external devices and mobile apps, without worrying that fraudulent data from prankers could ruin our analysis.

### When should you use Event Hubs:

Use Event Hubs when you need these features:

* Support authenticating a large number of publishers.
* Save a stream of events to Data Lake or Blob storage.
* Aggregation or analytics on your event stream.
* Reliable messaging or resiliency.

Event Hubs lets you build a big data pipeline capable of processing millions of events per second with low latency. It can handle data from concurrent sources and route it to a variety of stream-processing infrastructures and analytics services. It enables real-time processing and supports repeated replay of stored raw data.

## SUMMARY - How to choose a communications technology

1. Is the communication a message?

(Does the sending component expect the communication to be processed in a particular way by the destination component?)

If so, consider using Service Bus or Storage Queues.

### **Does a message have more than one destination?**

If so, use a Service Bus Topic.

### Do you need any special features?

### (E.g., Transactional messages, At most/least once guarantee, role-based access, messages larger than 64 KB etc.)

### (Your queue size will NOT grow larger than 80 GB.)

If so, use a Service Bus Queue.

### Are your requirements simple?

### (Audit trail of all messages, Supports unlimited queue size, track progress, code quickly)

(max message size is 64 KB)

If so, use a Storage Queue.

1. Is the communication an event?

If so, consider using Event Grid or Event Hubs.

### Is there high-flow stream of communications?

If so, consider using Event Hubs.

Otherwise, use Event Grid.

# Azure Service Bus

You have identified the following scenarios for message exchange between the mobile app and the web service:

1. Messages that relate to individual sales must be sent to the web service instance in the user's region.
2. Messages that relate to sales performance must be sent to all instances of the web service.

You have decided to implement a Service Bus queue for the first use case and a Service Bus topic for the second use case.

## Create a Service Bus namespace

Azure portal -> Create a resource -> Service Bus -> Create->

| **Setting** | **Value** | **Description** |
| --- | --- | --- |
| Subscription | Concierge subscription | The subscription in which this new app is created. |
| Resource group | [Sandbox resource group name] | The name of the resource group in which to create your Service Bus namespace. |
| Namespace name | [Globally unique name] | Enter a name that is unique in Azure. If you want to use the format *salesteamapp*<*Company*><*year*>, your namespace name would look like the example *salesteamappContoso2022*. |
| Location | Select from the dropdown | Choose from the free *sandbox regions* listed after this table. |
| Pricing tier | Standard | The recommended pricing tier for this exercise. |

-> Review + create -> Create -> Go to resource.

### Get Connection String for the storage account by following steps:

-> Service Bus namespace -> Left menu -> Shared access policies

-> RootManageSharedAccessKey -> Primary connection string -> Copy to clipboard.

## Create a Service Bus queue

Service Bus Namespace -> Left menu Entities -> Queues -> [+ Queue] button

-> Create queue pane -> “Name” = salesmessages -> Create.

## Create a Service Bus topic

Service Bus Namespace -> Left menu Entities -> Topics -> [+ Topic] button

-> Create topic pane

-> “Name” = salesperformancemessages

“Enable partitioning” = checked -> select Create.

## Create subscriptions for Service Bus topic

Salesperformancemessages -> [+ Subscription] button -> Create subscription pane

-> “Name” = Americas, “Max delivery count” = 100 -> Create.

Salesperformancemessages -> [+ Subscription] button -> Create subscription pane

-> “Name” = EuropeAndAsia, “Max delivery count” = 100 -> Create.

## Write code to send and receive messages by using a queue / a topic

### NuGet package required:

**Azure.Messaging.ServiceBus**

[Find the code here](https://github.com/AjinkyaApte88/General/tree/main/mslearn-service-bus/implement-message-workflows-with-service-bus/src/start).

**Service Bus queues and topics are excellent tools you can use to increase the resilience of communications within a distributed application. By acting as temporary storage locations, Service Bus queues and topics remove the requirement for direct communication between components, and they smoothly handle peaks in demand.**

**Consider using Service Bus queues and topics when you have a component that can communicate with another component in a loosely coupled configuration.**

# Azure Queue storage

Direct communication between the components of a distributed application can be problematic because it might be disrupted when network bandwidth is low or when demand is high.

We've seen this in our system: the web portal calls a web service, which works great if the service responds in a timely manner. High traffic causes problems and so the plan is to use a queue to eliminate the direct link between the front-end apps and your middle-tier web service.

## Create a storage account

| **Parameter** | **Value** |
| --- | --- |
| Name | Sets the name. Remember that storage accounts use the name to generate a public URL - so it must be unique. In addition, the account name must be between 3 and 24 characters, and be composed of numbers and lowercase letters only. We recommend you use the prefix **articles** with a random number suffix, but you can use whatever you like. |
| Resource group | Supplies the **Resource Group**. Use *[sandbox resource group name]* as the value. |
| Account Type | Sets the **Storage Account type**: *StorageV2* to create a general-purpose V2.account. |
| Redundancy | Sets the **Replication and Storage type**. It defaults to *Standard\_RAGRS (Global Redundancy)*.  Let's use *Standard\_LRS* (locally redundant within the datacenter). |
| location | Sets the **Location** independent of the resource group owner. It's optional, but you can use it to place the queue in a different region than the resource group. Place it close to you. |

-> Review + create -> Create -> Go to resource.

### Get Connection String for the storage account by following steps:

Azure storage account -> Left menu -> Access keys -> Connection String -> Show

-> Copy to clipboard

## Programmatically create and access a queue

### NuGet package required:

**Azure.Storage.Queues**

[Find the code here](https://github.com/AjinkyaApte88/General/tree/main/mslearn-storage-queues/start).

**Using the Azure.Storage.Queues package for .NET can help to make a distributed application more reliable and resilient to failures and periods of high demand.**