**Create serverless applications**

**Choose the best Azure service to automate your business processes**

* Logic Apps
* Microsoft Power Automate
* WebJobs
* Azure Functions

**Design-first technologies**

They both include user interfaces in which you can draw out the workflow

* **Logic Apps** is a service within Azure that you can use to automate, orchestrate, and integrate disparate components of a distributed application. By using the design-first approach in Logic Apps, you can draw out complex workflows that model complex business processes.

You can create or edit a workflow in JSON

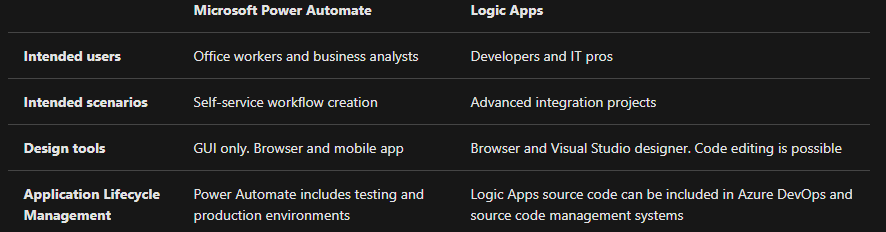
One reason why Logic Apps is so good at integration is that over 200 connectors are included. You can create your own connector if your system exposes a REST API.

* **Microsoft Power Automate** is a service that you can use to create workflows even when you have no development or IT Pro experience. You can create workflows that integrate and orchestrate many different components by using the website or the Microsoft Power Automate mobile app.

There are four different types of flow that you can create:

* + **Automated** a flow that is started by a trigger from some event.
  + **Button**
  + **Scheduled**
  + **Business process** a flow that models a business process such as the stock ordering process or the complaints procedure. The flow process can have: notification to required people; with their approval recorded; calendar dates for steps; and recorded time of flow steps.

*Under the hood, Microsoft Power Automate is built on Logic Apps. This fact means that Power Automate supports the same range of connectors and actions*



**Code-first technologies**

This is the case when you need more control over the performance of your workflow or need to write custom code as part of the business process.

* **WebJobs and the WebJobs SDK** WebJobs are a part of the Azure App Service that you can use to run a program or script automatically.  
  There are two kinds of WebJob:
  + **Continuous** run in a continuous loop. For example, you could use a continuous WebJob to check a shared folder for a new photo.
  + **Triggered** run when you manually start them or on a schedule.

The SDK includes a range of classes, such as **JobHostConfiguration** and **HostBuilder**, which   
reduce the amount of code required to interact with the Azure App Service. The WebJobs SDK only supports C# and the NuGet package manager.

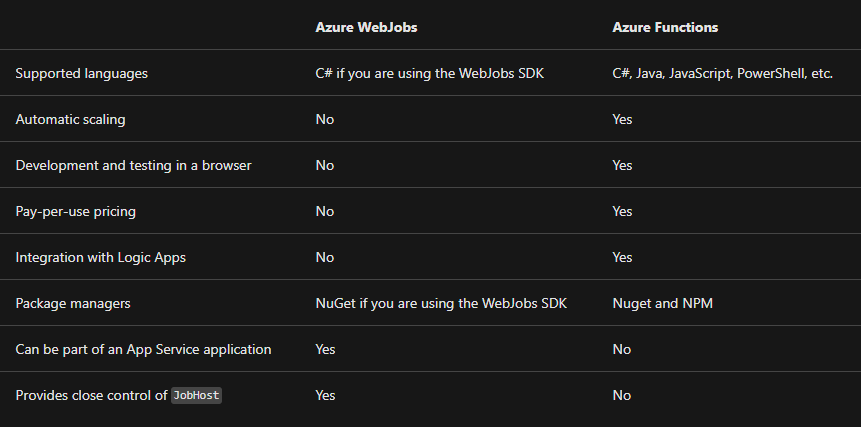
* **Azure Functions** is a simple way for you to run small pieces of code in the cloud, without having to worry about the infrastructure required to host that code.

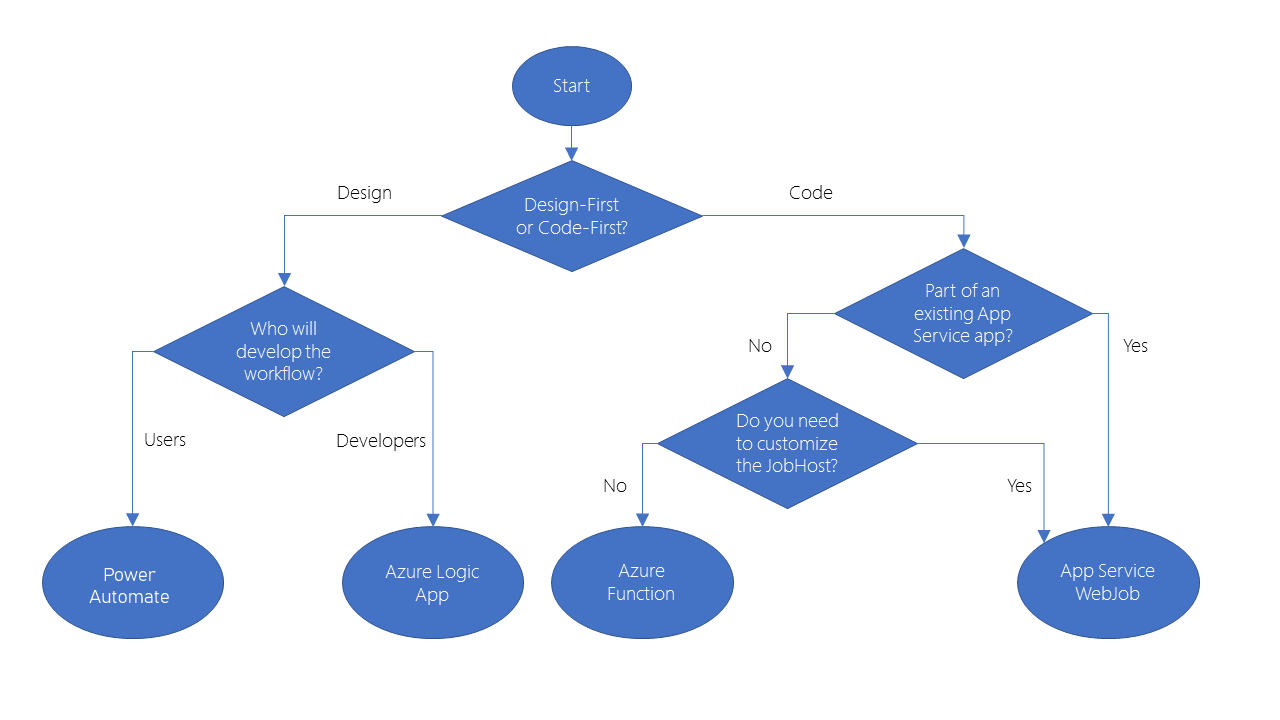
*In addition, with the consumption plan option, you only pay for the time when the code runs.* Azure automatically scales your function in response to the demand from users.

When you create an Azure Function, you can start by writing the code for it in the portal.

Alternatively, if you need source code management, you can use GitHub or Azure DevOps Services.

* + **HTTPTrigger**
  + **TimerTrigger**
  + **BlobTrigger**
  + **CosmosDBTrigger**



**How to choose a service**

**Create serverless logic with Azure Functions  
  
Execution time**

By default, functions have a timeout of 5 minutes. This timeout is configurable to a maximum of 10 minutes. If your function requires more than 10 minutes to execute, you can host it on a VM.   
If your service is initiated through an HTTP request and you expect that value as an HTTP response, the timeout is further restricted to 2.5 minutes. Finally, there's also an option called **Durable Functions** that allows you to orchestrate the executions of multiple functions without any timeout.

**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.  
While scaling, only one function app instance can be created every 10 seconds, for up to 200 total instances. Keep in mind, each instance can service multiple concurrent executions, so there is no set limit on how much traffic a single instance can handle. Different types of triggers have different scaling requirements, so research your choice of trigger and investigate its limits.

**What is a function app?**

Functions are hosted in an execution context called a function app. You define function apps to logically group and structure your functions and a compute resource in Azure.

**Choose a service plan**

Function apps may use one of two types of service plans. The first service plan is the **Consumption service plan.**The plan that you choose when using the Azure serverless application platform. The Consumption service plan provides automatic scaling and bills you when your functions are running. The Consumption plan comes with a configurable timeout period for the execution of a function. By default, it is 5 minutes, but may be configured to have a timeout as long as 10 minutes.  
**Azure App Service plan** allows you to avoid timeout periods by having your function run continuously on a VM that you define. When using an App Service plan, you are responsible for managing the app resources the function runs on, so this is technically not a serverless plan. However, it may be a better choice if your functions are used continuously or if your functions require more processing power or execution time than the Consumption plan can provide.

**Storage account requirements**

When you create a function app, it 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.

**Triggers**

Functions are event driven, which means they run in response to an event.  
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. |

### Bindings

Bindings are a declarative way to connect data and services to your function. Bindings know how to talk to different services, which means you don't have to write 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 special type of input binding that has the additional capability of initiating execution.

### Secure HTTP triggers

HTTP triggers let you use API keys to block unknown callers by requiring the key to be present on each request. When you create a function, you select the authorization level. By default, it's set to Function, which requires a function-specific API key, but it can also be set to Admin to use a global "master" key, or Anonymous to indicate that no key is required.

Because we specified Function when we created this function, we will need to supply the key when we send the HTTP request. You can send it as a query string parameter named code, or as an HTTP header (preferred) named x-functions-key.

**Execute an Azure Function with triggers**

**What is a timer trigger?**

A timer trigger is a trigger that executes a function at a consistent interval. To create a timer trigger, you need to supply two pieces of information.

* **A Timestamp** parameter name, which is simply an identifier to access the trigger in code.
* **A Schedule**, which is a CRON expression that sets the interval for the timer.

**What is a CRON expression?**

A CRON expression is a string that consists of six fields that represent a set of times. The order of the six fields in Azure is: {second} {minute} {hour} {day} {month} {day of the week}.

For example, a CRON expression to create a trigger that executes every five minutes looks like:

0 \*/5 \* \* \* \*

| **To build a CRON expression, you need to have a basic understanding of some of the special characters.** | | |
| --- | --- | --- |
| **Special character** | **Meaning** | **Example** |
| \* | Selects every value in a field | An asterisk "\*" in the day of the week field means *every* day. |
| , | Separates items in a list | A comma "1,3" in the day of the week field means just Mondays (day 1) and Wednesdays (day 3). |
| - | Specifies a range | A hyphen "10-12" in the hour field means a range that includes the hours 10, 11, and 12. |
| / | Specifies an increment | A slash "\*/10" in the minutes field means an increment of every 10 minutes. |

When you put all the fields together, the expression is read as "on the first second, of every fifth minute of every hour, of every day, of every month".

**Execute an Azure function with an HTTP request**

**What is an HTTP trigger Authorization level?**

An HTTP triggerAuthorization level is a flag that indicates if an incoming HTTP request needs an API key for authentication reasons.

**There are three Authorization levels:**

* Function
* Anonymous
* Admin

The Function and Admin levels are "key" based. To send an HTTP request, you must supply a key for authentication. There are two types of keys: function and host. The difference between the two keys is their scope. **Function keys are specific to a function. Host keys apply to all functions inside the function app.** If your Authorization level is set to Function, you can use either a function or a host key. If your Authorization level is set to Admin, you must supply a host key.  
The Anonymous level means that there's no authentication required. We use this level in our exercise.

**Execute an Azure function when a blob is created**

**What is Azure Blob storage?**

Azure Blob storage is an object storage solution that's designed to store large amounts of unstructured data. For example, Azure Blob storage is great at doing things like:

* Storing files
* Serving files
* Streaming video and audio
* Logging data

There are three types of blobs: block blobs, append blobs, and page blobs.

* **Block blobs** are the most common type. They allow you to store text or binary data efficiently.
* **Append blobs** are like block blobs, but they're designed more for append operations like creating a log file that's being constantly updated.
* **Page blobs** are made up of pages and are designed for frequent random read and write operations.

**How to create a blob trigger**

One setting that you'll want to look at is the Path. The Path tells the blob trigger where to monitor to see if a blob is uploaded or updated. By default, the Path value is:

samples-workitems/{name}

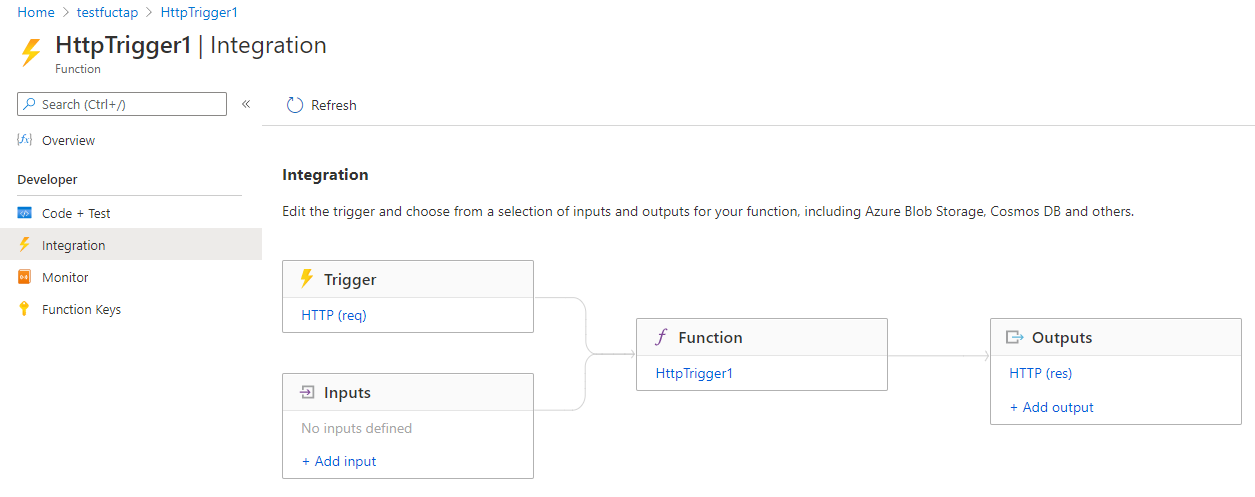
Let's break down this concept into two pieces: samples-workitems and {name}. The first part, samples-workitems, represents the blob container that the trigger monitors. The second part, {name} means that every type of file will cause the trigger to invoke the function. The function is invoked because there's no filter. For example, we could make the trigger invoke the function only when a PNG file is added by using syntax like:

samples-workitems/{name}.png

The last significant piece of information with this concept is the text name. The name represents a parameter in your Azure function that receives the name of the added file.

**True or false: an Azure Function can have multiple triggers associated with it?  
R:** *Every Azure Function must have exactly one trigger associated with it. If you want to use multiple triggers, you must create multiple functions.*

**Chain Azure Functions together using input and output bindings**



*You can see that we can't add more than one trigger. In fact, to change the trigger for our function, we would have to first delete the trigger, and create a new one. However, the Inputs and Outputs sections of this page display a plus sign (+) to add more bindings so we can accept more than one input value, and emit more than one output value.*

**What is a binding?**

Provide a declarative way to connect to data from within your code.   
This is powerful because you can connect to your data sources without having to code specific connection logic (like database connections or web API interfaces).

**Types of bindings**

* **Input binding** - An input binding is a connection to a data source. Our function can read data from these inputs.
* **Output binding** - An output binding is a connection to a data destination. Our function can write data to these destinations.

**Types of supported bindings**

* Blob Storage
* Azure Service Bus Queues
* Azure Cosmos DB
* Azure Event Hubs
* External Files
* External Tables
* HTTP endpoints

*A binding type can be used as an input, an output or both.*

**Binding properties**

Three properties are required in all bindings. You may have to supply additional properties based on the type of binding and storage you are using.

* **Name** - Defines the function parameter through which you access the data. For example, in a queue input binding, this is the name of the function parameter that receives the queue message content.
* **Type** - Identifies the type of binding, i.e., the type of data or service we want to interact with.
* **Direction** - Indicates the direction data is flowing, i.e., is it an input or output binding?

Additionally, most binding types also need a fourth property:

* **Connection** - Provides the name of an app setting key that contains the connection string. Bindings use connection strings stored in app settings to keep secrets out of the function code. This makes your code more configurable and secure.

{  
 "name": "headshotBlob",  
 "type": "blob",  
 "path": "thumbnail-images/{filename}",  
 "connection": "HeadshotStorageConnection",  
 "direction": "in"  
 }

The path property is required when using the Blob trigger, and should be provided in the style shown here, with curly braces around the filename portion of the path. This creates a **binding expression** that allows you to reference the blob's name in other bindings and in your function's code.

**Write data with output bindings**

As with input bindings, there are multiple types of output bindings. However not all types support both input and output. You'll use them anytime you want to send or store data. Here, we'll look at the types that support output bindings and when to use them.

**Output binding types**[**https://docs.microsoft.com/en-us/learn/modules/chain-azure-functions-data-using-bindings/6-write-data-with-output-bindings-portal-lesson**](https://docs.microsoft.com/en-us/learn/modules/chain-azure-functions-data-using-bindings/6-write-data-with-output-bindings-portal-lesson)

**Combining input and output bindings**

It's possible to apply multiple bindings to a single function. This allows you to define both input and output bindings, and the input and output can even be the same binding type.

**Input binding types**[**https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings?tabs=csharp#supported-bindings**](https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings?tabs=csharp#supported-bindings)

**What is a binding expression?**

A binding expression is specialized text in function.json, function parameters, or code that is evaluated when the function is invoked to yield a value. For example, if you have a Service Bus Queue binding, you could use a binding expression to get the name of the queue from App Settings.

**Types of binding expressions**

* App settings
* Trigger file name
* Trigger metadata
* JSON payloads
* New GUID
* Current date and time

Most expressions are identified by wrapping them in curly braces. However, app setting binding expressions are wrapped in percent signs rather than curly braces. For example if the blob output binding path is %Environment%/newblob.txt and the Environment app setting value is Development, a blob will be created in the Development container.

**Add an Azure Cosmos DB input binding**

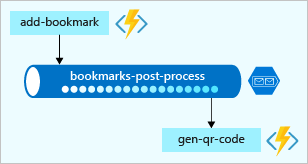
In the Document ID field, enter {id}.

This syntax is known as a binding expression. The function is triggered by an HTTP request that uses a query string to specify the ID to look up.

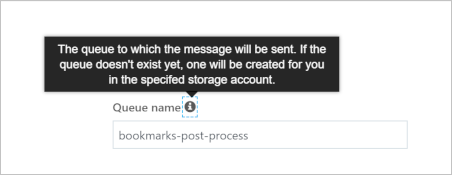
An incoming HTTP request triggers the function, and an id query parameter is passed to the Azure Cosmos DB input binding. If the database finds a document that matches this ID, the bookmark parameter will be set to the located document. In that case, you construct a response that contains the URL value found in the bookmarked document. If no document is found matching this key, you would respond with a payload and status code that tells the user the bad news.

**Add an Azure Queue Storage output binding**

Azure Queue storage is a service for storing messages that can be accessed from anywhere in the world. The size of a single message can be as much as 64 KB, and a queue can contain millions of messages - up to the total capacity of the storage account in which it is defined. T



*The only task you performed was to create a queue binding. You never created the queue explicitly. You are witnessing the power of bindings! As the following notification declares, the queue is automatically created for you if it doesn't exist.*



In this exercise, we expanded your knowledge of bindings to output bindings, writing data to your Azure Cosmos DB. We went further and added another output binding to post messages to an Azure queue. This demonstrates the true power of bindings to help you shape and move data from incoming sources to a variety of destinations. We haven't written any database code or had to manage connection strings ourselves. Instead, we configured bindings declaratively and let the platform take care of securing connections, scaling our function, and scaling our connections.

**Create a long-running serverless workflow with Durable Functions**

**Durable Functions**

Is an extension of Azure Functions that enables you to perform long-lasting, stateful operations in Azure. Azure provides the infrastructure for maintaining state information. You can use Durable Functions to orchestrate a long-running workflow. Using this approach, you get all the benefits of a serverless hosting model, while letting the Durable Functions framework take care of activity monitoring, synchronization, and runtime concerns.

*Durable Functions is an extension of Azure Functions****. Whereas Azure Functions operate in a stateless environment, Durable Functions can retain state between function calls****. This approach enables you to simplify complex stateful executions in a serverless-environment.*

***Some benefits of using Durable Functions include:***

* They enable you to write event driven code. A durable function can wait asynchronously for one or more external events, and then perform a series of tasks in response to these events.
* You can chain functions together. You can implement common patterns such as fan-out/fan-in, which uses one function to invoke others in parallel, and then accumulate the results.
* You can orchestrate and coordinate functions, and specify the order in which functions should execute.
* The state is managed for you. You don't have to write your own code to save state information for a long-running function.

**An orchestration function provides these extra benefits:**

* You can define the workflows in code. You don't need to write a JSON description or use a workflow design tool.
* Functions can be called both synchronously and asynchronously. Output from the called functions is saved locally in variables and used in subsequent function calls.
* 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**

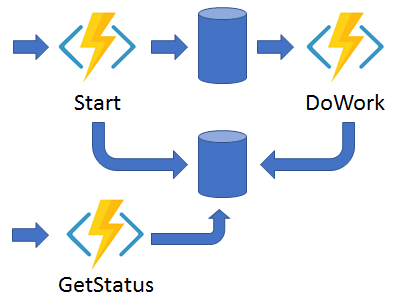
* **Client** functions are the entry point for creating an instance of a Durable Functions orchestration. They can run in response to an event from many sources, such as a new HTTP request arriving, a message being posted to a message queue, an event arriving in an event stream. You can write them in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

**Function chaining** - 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 then waits for all the functions to finish. The results of the parallel executions can be aggregated or used 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.



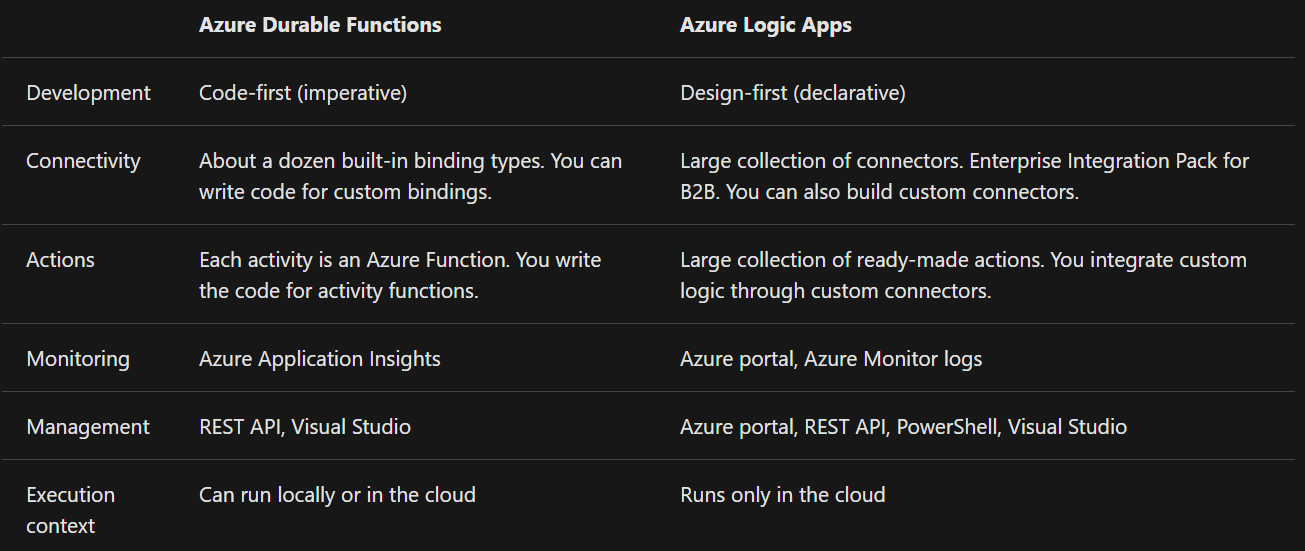
**Monitor -** This pattern implements a recurring process in a workflow, possibly looking for a change in state. For example, you could use this pattern to poll until specific conditions are met.



**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.



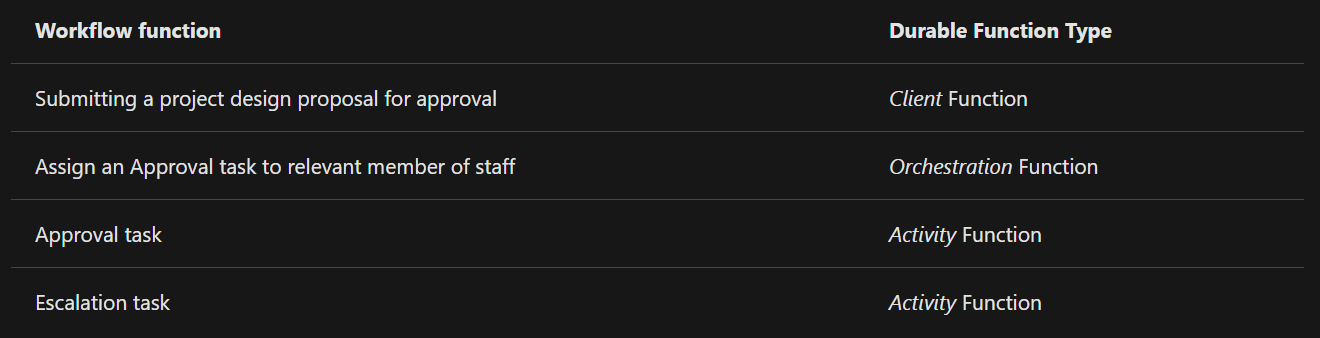
**Comparison with Logic Apps**



**Description of the design approval process**

1. A project design is submitted.
2. An approval task is allocated to a manager, so they can review the project design proposal.
3. The project design proposal is rejected or approved.
4. 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.**

****

**How to control long running tasks using durable timers & How to add an escalation path based on the timer**

**Timers in Durable Functions**

Durable Functions provides timers for use in the orchestrator functions, which you can use to implement delays or set up timeouts for asynchronous actions. You should use durable timers in orchestrator functions instead of the setTimeout() and setInterval() functions.

You create a durable timer by calling the createTimer() method of the DurableOrchestrationContext. This method returns a task that resumes on a specified date and time.

**Using timers for delay**

*You should always use currentUtcDateTime to obtain the current date and time, instead of Date.now or Date.UTC.*

*Durable Functions enables you to implement long-running workflows without requiring that you maintain state information manually. Azure provides the infrastructure in which Durable Functions run. You focus on the logic for the functions that perform the tasks in your workflow.*

*In this module we learned about Durable Functions and how to orchestrate our simple design proposal approvals process. Using durable timers, we can add an escalation path to our workflows, especially for those long-running, indeterminate tasks.*

**Develop, test, and publish Azure Functions by using Azure Functions Core Tools**

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

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

The Azure Functions Core Tools let you develop and run functions on your local computer from the command line.

**What are the Azure Functions Core Tools?**

The Azure Functions Core Tools are a set of command-line tools that you can use to develop and test Azure Functions on your local computer.

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

**Function apps and functions projects**

Every function published to Azure belongs to a function app: a collection of functions that are published together into the same environment. All of the functions in an app share a common set of configuration values, and must all be built for the same language runtime. Each function app is an Azure resource that can be configured and managed independently.

When you develop functions locally, you work within a functions project: a folder that contains the code and configuration files that define your functions. 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.

To get started developing, you need to create a functions project folder that's organized correctly. Every new function you add to the project requires additional code and configuration that must be complete and correctly structured, or your functions will not be able to run. If you wanted to, you could become familiar with the names and contents of the files needed in a functions project folder and create them yourself, but doing so would be time-consuming and error-prone.

With the Azure Functions Core Tools, you'll never need to do this! You can use the Core Tools to generate function projects and functions from scratch.

**When you create a new functions project two most critical project files are always present:**

* **host.json** stores runtime configuration values, such as logging options, for the function app. The settings stored in this file are used both when running functions locally and in Azure.
* **local.settings.json** stores configuration values that only apply to the function app when it is run locally with the Core Tools. This file contains two kinds of settings:
  + **local runtime settings** used to configure the local functions runtime itself
  + **custom application settings** which you can add and configure based on your app's needs and can be accessed and used by all the functions in the app.

**Run functions locally**

Functions aren't programs that can be run on their own: they must be hosted by the functions host. The host is what powers everything outside of your function code: it loads configuration, listens for triggers and HTTP requests, starts the worker process for the language your functions are written in, writes log output and more. In Azure, function apps run the function host automatically when they

You can use the Core Tools to run your own instance of the functions host and try out your functions locally before you publish them. By running your functions before publishing them, you can make sure your configuration and code loads correctly and test out your functions by making real HTTP calls to them without the need for Azure resources.

**Exercise - Create a function locally by using the Core Tools**

mkdir ~/loan-wizard

cd ~/loan-wizard

func init

func new

code .

***Replace the full contents of index.js***

func start

Ctrl+C

func start &> ~/output.txt &

curl "http://localhost:7071/api/simple-interest" -w "\n"

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

pkill func

code ~/output.txt

**Create a function app**

Before you can use the Core Tools to publish a project, you need to create a function app in Azure. This is not a capability of the Core Tools: creating function apps is one of the responsibilities of the Azure management tools, which include the Azure portal, Azure CLI and Azure PowerShell.

*If you already have a local functions project you want to publish, make sure to create the function app with the same language runtime. If you try to deploy a project to an app with a different runtime, publishing will halt with an error.*

**Publish to Azure**

To publish a functions project to Azure, run **func azure functionapp publish <app\_name>** from the functions project folder. **<app\_name>** is the name of the target function app in Azure, not the name of your project folder, which can be different.

*The Core Tools don't ask you to sign in to Azure. Instead, they access your subscriptions and resources by loading your session information from the Azure CLI or Azure PowerShell. If you don't have an active session in one of those tools, publishing will fail. It's possible to publish from the Core Tools without the Azure CLI or Azure PowerShell, but it's much easier if you do have them, and we recommend you install one or the other and log in before trying to publish.*

**Things to know**

* The Core Tools do not validate or test your functions code during publishing.
* When you publish, any functions already present in the target app are stopped and deleted before the contents of your project are deployed. You can't combine functions from multiple projects into one app by publishing them in sequence - all of the functions you want in the app must be in one project.
* Publishing to Azure does not create any kind of relationship between the local project and the target function app. You can publish a single functions project to multiple function apps. You can also re-publish a project to the same app repeatedly as you work on your code.
* The invocation URLs displayed after you publish may include a code parameter in the query string, as in the screenshot above. By default, HTTP functions created by the Core Tools are configured with an authorization level of function, meaning they require callers to provide a secret key in the request headers or query string. The Core Tools includes the key in the query string of the displayed URL for your convenience.

**Publish a function to Azure by using the Core Tools**

RESOURCEGROUP=learn-1a731473-8658-4cd9-b51b-57b5612f8a15

STORAGEACCT=learnstorage$(openssl rand -hex 5)

FUNCTIONAPP=learnfunctions$(openssl rand -hex 5)

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 2

1. The three lines at the top create shell variables with values that we use repeatedly in the following commands. For resource group, we specify the group created for you by the sandbox. The storage account and function app names include $(openssl rand -hex 5), which generates a random 5-character string, to ensure that the names meet the requirement of being globally unique.
2. az storage account create creates an Azure storage account that will be used by the function app. A storage account is a separate Azure resource that needs to be created before the function app can be created.
3. az functionapp create creates the function app. Our new app uses the node (JavaScript) runtime, and runs on the serverless, pay-as-you-go consumption billing plan.

cd ~/loan-wizard

func azure functionapp publish "$FUNCTIONAPP"

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

[**https://docs.microsoft.com/en-us/learn/modules/develop-test-deploy-azure-functions-with-visual-studio/2-develop-and-test-azure-functions-locally**](https://docs.microsoft.com/en-us/learn/modules/develop-test-deploy-azure-functions-with-visual-studio/2-develop-and-test-azure-functions-locally)

**Monitor GitHub events by using a webhook with Azure Functions**

**Webhooks** offer a lightweight mechanism for apps to be notified by another service when something of interest happens via an HTTP endpoint. You can use a webhook to trigger an Azure function, and then analyze the message, to determine what happened and how to respond.

In GitHub, webhooks can be set up on an organization or a specific repository. The webhook will be triggered each time one or more subscribed events occurs. For example, the **Gollum** event allows you to listen for wiki updates; specifically creation and updates for a wiki page.

**Setting up a webhook**

Setting up a webhook is a two-step process. You specify how you want your webhook to behave through GitHub and what events it should listen to. Then you set up your function in Azure Functions to receive and manage the payload received from the webhook.

**Webhooks require a couple of configuration options before you can use them. We'll go through each of these settings next.**

* **Payload URL** The payload URL is the URL of the server that will receive the webhook POST requests. Each event type has a specific payload format. That payload contains information about the event that triggered the webhook.
* **Content Type** Webhooks can be delivered using two different content types:
  + The application/json content type delivers the JSON payload directly as the body of the POST request.
  + The application/x-www-form-urlencoded content type sends the JSON payload as a form parameter, called payload.
* **Events** Are at the center of webhooks. Events occur whenever actions are taken in the repository. When the event occurs, the webhook fires off and calls the URL that you specify, sending along the payload and event information to your URL. For example, to respond whenever an issue is raised in a repository
  + Click Let me select individual events
  + Select Issues.
  + Make sure you select Active to receive issue events for triggered webhooks.

To listen for updates to the wiki for the repository, select the Wiki checkbox; this event is the Gollum event mentioned earlier.

<https://docs.microsoft.com/en-us/learn/modules/monitor-github-events-with-a-function-triggered-by-a-webhook/5-exercise-setup-webhook-for-github-repo>

**Secure Webhook payloads with a secret**

Once your function is configured to receive payloads, it will listen for any payload sent to the endpoint you configured. For security reasons, you might want to limit requests to those coming from GitHub. There are a few ways to go about this. For example, you could opt to approve requests from GitHub's IP address. An easier method is to set up a secret token and validate the request using this token.

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

<https://docs.microsoft.com/en-us/learn/modules/automatic-update-of-a-webapp-using-azure-functions-and-signalr/>

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

**The Azure API Management (APIM)**

Is a fully managed cloud service that you can use to publish, secure, transform, maintain, and monitor APIs. 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. APIM enables you to create and manage modern API gateways for existing backend services.

**APIM Consumption Tier**

The consumption tier uses the same underlying service components as the previous tiers, but employs an entirely different architecture based on shared, dynamically allocated resources. The consumption tier aligns perfectly with serverless computing models; there is no infrastructure to manage, no idle capacity, high-availability, automatic scaling, and usage-based pricing, all of which make it an especially good choice for solutions that involve exposing serverless resources as APIs.

**How does API Management help?**

* Client apps are coupled to the API expressing business logic, not the underlying technical implementation with individual microservices. You can change the location and definition of the services without necessarily reconfiguring or updating the client apps.
* API Management acts as an intermediary. It forwards requests to the right microservice, wherever it is located, and returns responses to users. Users never see the different URIs where microservices are hosted.
* You can use API Management policies to enforce consistent rules on all microservices in the product. For example, you can transform all XML responses into JSON, if that is your preferred format.
* Policies also enable you to enforce consistent security requirements.

**Connect your services together**

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

* Azure Storage queues
* Azure Event Hubs
* Azure Event Grid
* Azure Service Bus.

**Choose whether to use messages or events**

**What is a message?**

* A message contains raw data, produced by one component, that will be consumed by another component.
* A message contains the data itself, not just a reference to that data.
* The sending component expects the message content to be processed in a certain way by the destination component. The integrity of the overall system may depend on both sender and receiver doing a specific job.

**What is an event?**

Events are lighter weight than messages, and are most often used for broadcast communications. The components sending the event are known as publishers, and receivers are known as subscribers.

With events, receiving components will generally decide in which communications they are interested, and will "subscribe" to those events. The subscription is managed by an intermediary, like **Azure Event Grid or Azure Event Hubs.** When publishers send an event, the intermediary will route that event to interested subscribers. This pattern is known as a **"publish-subscribe architecture."** It's not the only way to deal with events, but it is the most common.

* An event is a lightweight notification that indicates that something happened.
* The 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**

* Events are more likely to be used for broadcasts and are often ephemeral, meaning a communication might not be handled by any receiver if none is currently subscribing.
* Messages are more likely to be used where the distributed application requires a guarantee that the communication will be processed.

**Does the sending component expect the communication to be processed in a particular way by the destination component?  
R:** If the answer is yes, choose to use a message. If the answer is no, you may be able to use events.

**Choose a message-based delivery with queues**

**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 subscriptio**n. 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.

**Message delivery guarantees**

* At-Least-Once Delivery:
* At-Most-Once Delivery:
* First-In-First-Out (FIFO):

**Transactional support**

Message transactions succeed or fail as a single unit - just like in the database world. If the credit card details message delivery fails, so will the order details message.

**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 256 KB.
* Queue size will not grow larger than 80 GB.
* 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 80 GB in size.
* Want to track progress for processing a message inside of the queue.

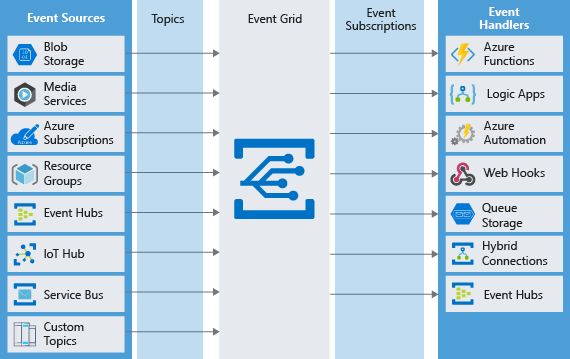
**Use Storage queues** when you want a simple and easy-to-code queue system. For more advanced needs, use **Service Bus queues**. If you have multiple destinations for a single message, but need queue-like behavior, use **Service Bus topics.**

**What is Azure Event Grid?**

Is a fully-managed **event routing service** running on top of Azure Service Fabric. 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.

There are several concepts in Azure Event Grid that connect a source to a subscriber:

* **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:



| **WHAT IS AN EVENT?** | |
| --- | --- |
| **Field** | **Description** |
| **topic** | The full resource path to the event source. Event Grid provides this value. |
| **subject** | Publisher-defined path to the event subject. |
| **id** | The unique identifier for event. |
| **eventType** | 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. |
| **eventTime** | The time the event was generated based on the provider's UTC time. |
| **data** | Specific information that is relevant to the type of event. For example, an event about a new file being created in Azure Storage has details about the file, such as the lastTimeModified value. Or, an Event Hubs event has the URL of the Capture file. This field is optional. |
| **dataVersion** | The schema version of the data object. The publisher defines the schema version. |
| **metadataVersion** | The schema version of the event metadata. Event Grid defines the schema of the top-level properties. Event Grid provides this value. |

*Event Grid sends an event to indicate something has happened or changed. 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.*

**What is an event source?**

Event sources are responsible for sending events to Event Grid. Each event source is related to one or more event types. For example, Azure Storage is the event source for blob created events. IoT Hub is the event source for device created events.

Azure Event Hub has the concept of an event publisher which is often confused with the event source. A publisher to Event Hub is the user or organization that decides to send events to Event Grid. For example, Microsoft publishes events for several Azure services.

**What is an event topic?**

Event topics categorize events into groups. Topics are represented by a public endpoint and are where the event source sends events to.

For example, consider an application that sends events related to modifying user accounts and processing orders. It's unlikely any event handler wants both categories of events. Create two custom topics and let event handlers subscribe to the one that interests them. Event subscribers can filter for the event types they want from a specific topic.

* **System topics** are built-in topics provided by Azure services. You don't see system topics in your Azure subscription because the publisher owns the topics, but you can subscribe to them. To subscribe, you provide information about the resource you want to receive events from. As long as you have access to the resource, you can subscribe to its events.
* **Custom topics** are application and third-party topics. When you create or are assigned access to a custom topic, you see that custom topic in your subscription.

**What is an event subscription?**

Event Subscriptions define which events on a topic an event handler wants to receive. A subscription can also filter events by their type or subject, so you can ensure an event handler only receives relevant events.

**What is an event handler?**

An event handler (sometimes referred to as an event "subscriber") is any component (application or resource) that can receive events from Event Grid.

**Should you use Event Grid?**

* **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.

**What is 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.

**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." By default, events stay in the buffer for 24 hours before they automatically expire.

The buffers are called partitions because the data is divided amongst them. Every event hub has at least two partitions, and each partition has a separate set of subscribers.

**Choose Event Hubs if:**

* You need to support authenticating a large number of publishers.
* You need to save a stream of events to Data Lake or Blob storage.
* You need aggregation or analytics on your event stream.
* You need reliable messaging or resiliency.

Otherwise, if you need a simple event publish-subscribe infrastructure, with trusted publishers (for instance, your own web server), you should choose Event Grid.

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.**

**What is a queue?**

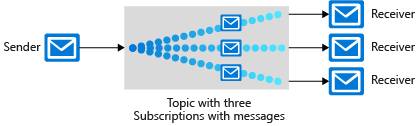
A queue is a simple temporary storage location for messages. A sending component adds a message to the queue. A destination component picks up the message at the front of the queue. Under ordinary circumstances, each message is received by only one receiver.



A queue responds to high demand like this without needing to add resources to the system. However, for messages that need to be handled relatively quickly, adding additional instances of your destination component can allow them to share the load. Each message would be handled by only one instance. This is an effective way to scale your entire application while only adding resources to the components that actually need it.

**What is a topic?**

A topic is similar to a queue but can have multiple subscriptions. This means that multiple destination components can subscribe to a single topic, so each message is delivered to multiple receivers. Subscriptions can also filter the messages in the topic to receive only messages that are relevant. Subscriptions provide the same decoupled communications as queues and respond to high demand in the same way. Use a topic if you want each message to be delivered to more than one destination component.



*Topics are not supported in the Basic pricing tier.*

**What is a relay?**

A relay is an object that performs synchronous, two-way communication between applications. Unlike queues and topics, it is not a temporary storage location for messages. Instead, it provides bidirectional, unbuffered connections across network boundaries such as firewalls. Use a relay when you want direct communications between components as if they were located on the same network segment but separated by network security devices.

**Key advantages of Service Bus queues include:**

* Supports larger messages sizes of 256 KB (standard tier) or 1MB (premium tier) per message versus 64 KB
* Supports both at-most-once and at-least-once delivery - choose between a very small chance that a message is lost or a very small chance it is handled twice
* Guarantees first-in-first-out (FIFO) order - messages are handled in the same order they are added (although FIFO is the normal operation of a queue, it is not guaranteed for every message)
* Can group multiple messages into a transaction - if one message in the transaction fails to be delivered, all messages in the transaction will not be delivered
* Supports role-based security
* Does not require destination components to continuously poll the queue

**Advantages of storage queues:**

* Supports unlimited queue size (versus 80-GB limit for Service Bus queues)
* Maintains a log of all messages

**Create a Service Bus namespace**

In Azure Service Bus, a namespace is a container, with a unique fully qualified domain name, for queues, topics, and relays. You must start by creating the namespace.

Each namespace has primary and secondary shared access signature encryption keys. To gain access to the objects within the namespace, a sending or receiving component must provide these keys when it connects.

**Connection strings and keys**

Source components and destination components both need two pieces of information to connect to a queue in a Service Bus namespace:

* **The location of the Service Bus namespace, also known as an endpoint:** The location is specified as a fully qualified domain name within the servicebus.windows.net domain. For example: pizzaService.servicebus.windows.net.
* **An access key:** Service Bus restricts access to queues, topics, and relays by requiring an access key.

**Set filters on subscriptions**

If you want to control that specific messages sent to the topic are delivered to particular subscriptions, you can place filters on each subscription in the topic. In the pizza application, for instance, our storefronts are running Universal Windows Platform (UWP) applications. Each store can subscribe to the "OrderCancellation" topic but filter for its own StoreId. We save internet bandwidth because we are not sending unnecessary messages to distant store locations. Meanwhile, the payment processing component subscribes to all our cancellation messages.

**Filters can be one of three types:**

* **Boolean Filters:** The TrueFilter ensures that all messages sent to the topic are delivered to the current subscription. The FalseFilter ensures that none of the messages are delivered to the current subscription. (This effectively blocks or switches off the subscription.)
* **SQL Filters:** A SQL filter specifies a condition by using the same syntax as a WHERE clause in a SQL query. Only messages that return True when evaluated against this subscription will be delivered to the subscribers.
* **Correlation Filters:** A correlation filter holds a set of conditions that are matched against the properties of each message. If the property in the filter and the property on the message have the same value, it is considered a match.

*For our StoreId filter, we could use a SQL filter. SQL filters are the most flexible, but they're also the most computationally expensive and could slow down our Service Bus throughput. In this case, we choose a correlation filter instead.*

**True or false: you can add a message to an Azure Service Bus queue that is 2 MB in size.  
R:** *An Azure Storage queue message must be smaller than 64 KB. A service bus queue can be up to 256 KB for standard tier, and 1MB for the premium tier.*