**What is serverless compute?**

Serverless compute 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. The cloud provider manages 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 focus on in this module.

**What is Azure Functions?**

Azure Functions is a serverless application platform. It allows 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#, and JavaScript. Support for NuGet and NPM is also included, so you can use popular libraries in your business logic.

# **Azure Functions Vs Azure Webjobs**

|  |  |
| --- | --- |
| **Azure Webjobs** | **Functions** |
| Write the code in console and deploy through visual studio. | Write code directly in portal, easily to deploy and test. |
| Runs as a background process in the context of App services. | Runs using Appservice/dynamic Hosting plan. |
| Scale the whole web. | New container will be created dynamically when required and have more possibilities with dynamic service plan. |
| If you have existing App service plan there is no additional cost. | Metered to nearest 100ms (GB-s) |
| VM is always required. | Server less computing and VM is not mandatory. |
| Can run locally, Deploy it easily to cloud. | Limited local compilation and not in matured state. |
| Host is customizable | Not supporting bringing your own host and customizing the own host. |

**Benefits of a serverless compute solution**

Serverless compute is a great option for hosting business logic code in the cloud. With serverless offerings such as Azure Functions, you can write your business logic in the language of your choice. You get automatic scaling, you have no servers to manage, and you are charged based on what is used — not on reserved time. Here are some additional characteristics of a serverless solution for you to consider.

**Avoids over-allocation of infrastructure**

Suppose you've provisioned VM servers and configured them with enough resources to handle your peak load times. When the load is light, you are potentially paying for infrastructure you're not using. Serverless computing helps solve the allocation problem by scaling up or down automatically, and you're only billed when your function is processing work.

**Stateless logic**

Stateless functions are great candidates for serverless compute; function instances are created and destroyed on demand. If state is required, it can be stored in an associated storage service.

**Event driven**

Functions are *event driven*. This means they run only in response to an event (called a "trigger"), such as receiving an HTTP request, or a message being added to a queue. You configure a trigger as part of the function definition. This approach simplifies your code by allowing you to declare where the data comes from (trigger/input binding) and where it goes (output binding). You don't need to write code to watch queues, blobs, hubs, etc. You can focus purely on the business logic.

**Functions can be used in traditional compute environments**

Functions are a key component of serverless computing, but they are also a general compute platform for executing any type of code. 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 to manage scaling, run on virtual networks, and even completely isolate your functions.

**Drawbacks of a serverless compute solution**

Serverless compute will not always be the appropriate solution to hosting your business logic. Here are a few characteristics of functions that may affect your decision to host your services in serverless compute.

**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. Additionally, 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**

The second characteristic is 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. In our elevator example, you would create a function app to host the escalator drive gear temperature service. There are a few decisions that need to be made to create the function app; you need to choose a service plan and select a compatible storage account.

### Choosing a service plan

Function apps may use one of two types of service plans. The first service plan is the **Consumption service plan**. This is 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.

The second plan is called the **Azure App Service plan**. This 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. On the Consumption service plan, this is also where the function code and configuration file are stored.

**Triggers**

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

The type of event that starts the function is called a **trigger**. You must configure a function with exactly one trigger.

Azure supports triggers for the following services.

| **Service** | **Trigger description** |
| --- | --- |
| Blob storage | Start a function when a new or updated blob is detected. |
| Cosmos DB | Start a function when inserts and updates are detected. |
| Event Grid | Start a function when an event is received from Event Grid. |
| HTTP | Start a function with an HTTP request. |
| Microsoft Graph Events | Start 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 | Start a function when a new item is received on a queue. The queue message is provided as input to the function. |
| Service Bus | Start a function in response to messages from a Service Bus queue. |
| Timer | Start 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.

Azure provides a [large number of bindings](https://docs.microsoft.com/azure/azure-functions/functions-triggers-bindings#supported-bindings) to connect to different storage and messaging services.

**A sample binding definition**

Let's look at an example of configuring a function with an input binding (trigger) and an output binding. Let's say we want to read data from Blob storage, process it in our function, and then write a message to a queue. You would configure an *input binding* of type *blob* and an *output binding* of type *queue*.

Bindings can be defined in the Azure portal, and are stored as JSON files, which you can also edit directly. The following JSON is sample definition of a trigger and binding for a function.

{

"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"

}

]

}

This example shows a function that is triggered by a message being added to a queue named **myqueue-items**. It then sends the return value of the function to the **outTable** table in Azure Table storage. This is a very simple example, we could change the output to be an email using a SendGrid binding, or put an event onto a Service Bus to notify some other component in our architecture, or even have multiple output bindings to push data to various services.

**Determine the best trigger for your Azure function**

An Azure function app doesn't do work until something tells it to execute. For example, we could create an Azure function to send out a reminder text message to our customers before an appointment. If we don't tell the function when it should run, our customers will never receive a message.

**Run function on a schedule**

It's common to execute a piece of logic at a set interval. Imagine you're a blog owner and you notice that your subscribers aren't reading your most recent posts. You decide that the best action is to send an email once a week to remind them to check your blog. You implement this logic using an Azure function app with a timer trigger to invoke your function weekly.

## 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.

1. A Timestamp parameter name, which is simply an identifier to access the trigger in code.
2. 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 \* \* \* \*

At first, this string may look confusing. We'll come back and break down these concepts when we have a deeper look at CRON expressions.

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. |

Now we'll go back to the original CRON expression example. Let’s try to understand it better by breaking it down field by field.

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

The **first field** represents seconds. This field supports the values 0-59. Because the field contains a zero, it selects the first possible value, which is one second.

The **second field** represents minutes. The value "\*/5" contains two special characters. First, the asterisk (\*) means "select every value within the field." Because this field represents minutes, the possible values are 0-59. The second special character is the slash (/), which represents an increment. When you combine these characters together, it means for all values 0-59, select every fifth value. An easier way to say that is simply "every five minutes."

The **remaining four fields** represent the hour, day, month, and weekday of the week. An asterisk for these fields means to select every possible value. In this example, we select "every hour of every day of every month."

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**

An HTTP request is a common operation on most platforms and devices. Whether it's a request to look up a word in a dictionary or to get the local weather, we send HTTP requests all the time. Azure Functions allows us to quickly create a piece of logic to execute when an HTTP request is received.

Here, you'll learn how to create and invoke an Azure function using an HTTP trigger. You'll also explore some of the customization options that are available.

## What is an HTTP trigger?

An HTTP trigger is a trigger that executes a function when it receives an HTTP request. HTTP triggers have many capabilities and customizations, including:

* Provide authorized access by supplying keys.
* Restrict which HTTP verbs are supported.
* Return data back to the caller.
* Receive data through query string parameters or through the request body.
* Support URL route templates to modify the function URL.

When you create an HTTP trigger, select a programming language, provide a trigger name, and select an Authorization level.

## What is an HTTP trigger Authorization level?

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

There are three Authorization levels:

1. Function
2. Anonymous
3. 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.

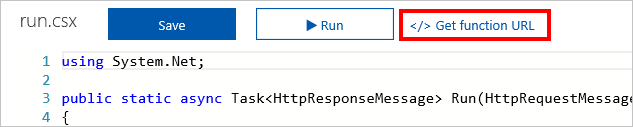
## How to create an HTTP trigger

Just like a timer trigger, you can create an HTTP trigger through the Azure portal. Inside your Azure function, you select **HTTP trigger** from the list of predefined trigger types. Then you enter the logic that you want to execute and make any customizations like restricting the use of certain HTTP verbs.

One setting that's important to understand is **Request parameter name**. This setting is a string that represents the name of the parameter that contains the information about an incoming HTTP request. By default, the name of the parameter is req.

## How to invoke an HTTP trigger

To invoke an HTTP trigger, you send an HTTP request to the URL for your function. To get this URL, go to the code page for your function and select the **Get function URL** link.



After you have the URL for your function, you can send HTTP requests. If your function receives data, remember that you can use either query string parameters or supply the data through the request body.

An HTTP trigger invokes an Azure function when it receives an HTTP request to its function URL. HTTP triggers allow you to receive data and return data back to the caller.

## What is a blob trigger?

A blob trigger is a trigger that executes a function when a file is uploaded or updated in Azure Blob storage. To create a blob trigger, you create an Azure Storage account and provide a location that the trigger monitors.

## How to create a blob trigger

Just like the other triggers we've seen so far, we create a blob trigger in the Azure portal. Inside your Azure function, select **Blob trigger** from the list of predefined trigger types. Then enter the logic to execute when a blob is created or updated.

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, I 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. For example, if I upload a file named resume.txt, my Azure function receives that value as a string through a parameter called name.

A blob trigger invokes an Azure function when it sees activity at a specific location in your Azure Storage blob account. You set the location to monitor by modifying the **Path** value in the Azure portal.

**Explore input and output binding types**

Accessing and processing data are key tasks in many software solutions. Consider some of these scenarios:

* You've been asked to implement a way to move incoming data from Blob storage to Azure Cosmos DB.
* You want to post incoming messages to a queue for processing by another component in your enterprise.
* Your service needs to grab gamer scores from a queue and update an online scoreboard.

All of these examples are about moving data. The data source and destinations differ from scenario to scenario, but the pattern is similar. You connect to a data source, and you read and write data. Azure Functions helps you integrate with data and services by using bindings.

**What is a binding?**

In Azure Functions, bindings provide a declarative way to connect to data from within your code. They make it easier to integrate with data streams consistently in a function. You can have multiple bindings providing access to different data elements. 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**

There are two kinds of bindings you can use with functions:

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

There are also triggers. Triggers are special types of input bindings that cause a function to execute. For example, an Azure Event Grid notification can be configured as a trigger. When an event occurs, the function will run.

**Types of supported bindings**

The *type* of binding defines where we are reading or sending data. There is a binding to respond to web requests and a large selection of bindings to interact directly with various Azure services as well as third-party services.

A binding type can be used as an input, an output or both. For example, a function can write to Azure Blob Storage output binding, but a blob storage update could trigger another function.

Some common binding types are listed below:

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

These types are just a sample. There are more, plus functions have an extensibility model to add more bindings.

**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.

1. **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.
2. **Type** Identifies the type of binding, i.e., the type of data or service we want to interact with.
3. **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:

1. **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.

**Create a binding**

Bindings are defined in JSON. A binding is configured in your function's configuration file, which is named *function.json* and lives in the same folder as your function code.

Let's examine a sample *input binding*:

...

{

"name": "headshotBlob",

"type": "blob",

"path": "thumbnail-images/{filename}",

"connection": "HeadshotStorageConnection",

"direction": "in"

},

...

To create this binding, we:

1. Create a binding in our *function.json* file.
2. Provide the value for the name variable. In this example, the variable holds the blob data.
3. Provide the storage type. In the preceding example, we are using Blob storage.
4. Provide the path, which specifies the container and the item name that goes in it. The path property is required for blobs.
5. Provide the connection string setting name defined in the application's settings file. It's used as a key to find the connection string to connect to your storage account.
6. Define the direction as in. It reads data from the blob.

Bindings are used to connect to data in your function. In this example, we used an input binding to connect user images to be processed by our function as thumbnails.

**Read data with input bindings**

To connect to a data source, you have to configure an input binding. This binding makes it possible to write minimal code to create a message. You don't have to write code for tasks such as opening a storage connection. The Azure Functions runtime and binding take care of those tasks for you.

## Input binding types

There are multiple types of input. However, not all types support both input and output. You'll use them anytime you want to ingest data of that type. Here, we'll look at the types that support input bindings and when to use them.

* **Blob storage** Blob storage bindings allow you to read from a blob.
* **Azure Cosmos DB** The Azure Cosmos DB input binding uses the SQL API to retrieve one or more Azure Cosmos DB documents and passes them to the input parameter of the function. The document ID or query parameters can be determined based on the trigger that invokes the function.
* **Microsoft Graph** Microsoft Graph input bindings allow you to read files from OneDrive, read data from Excel, and get auth tokens so you can interact with any Microsoft Graph API.
* **Mobile Apps** The Mobile Apps input binding loads a record from a mobile table endpoint and passes it into your function.
* **Table storage** You can read data and work with Azure Table storage.

To create a binding as an input, you must define the direction as in. The parameters for each type of binding may vary.

**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.

## Output binding types

* **Blob Storage** - You can use the blob output binding to write blobs.
* **Cosmos DB** - The Azure Cosmos DB output binding lets you write a new document to an Azure Cosmos DB database using the SQL API.
* **Event Hubs** - Use the Event Hubs output binding to write events to an event stream. You must have send permission to an event hub to write events to it.
* **HTTP** - Use the HTTP output binding to respond to the HTTP request sender. This binding requires an HTTP trigger and allows you to customize the response associated with the trigger's request. This can also be used to connect to web hooks.
* **Microsoft Graph** - Microsoft Graph output bindings allow you to write to files in OneDrive, modify Excel data, and send email through Outlook.
* **Mobile Apps** - The Mobile Apps output binding writes a new record to a Mobile Apps table.
* **Notification Hubs** - You can send push notifications with Notification Hubs output bindings.
* **Queue Storage** - Use the Azure Queue storage output binding to write messages to a queue.
* **Send Grid** - Send emails using SendGrid bindings.
* **Service Bus** - Use Azure Service Bus output binding to send queue or topic messages.
* **Table storage** - Use an Azure Table storage output binding to write to a table in an Azure Storage account.
* **Twilio** - Send text messages with Twilio.

To create a binding as an output, you must define the direction as out. The parameters for each type of binding may vary.

## 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.

**Install extensions in VS Code**

func extensions install

**Function Proxies**

Proxies are a nice addition to Azure Functions that give you a subset of features that an application gateway could provide for your function instances. It has request matching capabilities that let you inspect a specific route and send that request to a specific back-end. You can setup matches based on combinations of uri and HTTP method.

Proxies.json

{

"$schema": "http://json.schemastore.org/proxies",

"proxies": {

"MyEscalatorFnProxy": {

"matchCondition": {

"route": "/api/myesc",

"methods": [

"PUT"

]

},

"backendUri": "https://escalator-functions.azurewebsites.net/api/HttpTriggerSample",

"requestOverrides": {

"backend.request.method": "post"

}

},

"MyGoogleProxy": {

"matchCondition": {

"route": "/api/google"

},

"backendUri": "https://www.google.com"

}

}

}

# **Durable Functions**

Durable Functions is an extension of Azure Functions that lets you write stateful functions in a serverless environment. The extension manages state, checkpoints, and restarts for you.

**Benefits**

The extension lets you define stateful workflows using an orchestrator function, which can provide the following benefits:

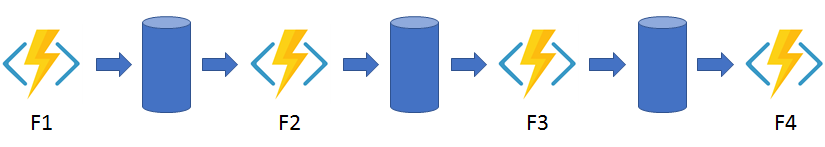
* You can define your workflows in code. No JSON schemas or designers are needed.
* Other functions can be called both synchronously and asynchronously. Output from called functions can be saved to local variables.
* Progress is automatically checkpointed when the function awaits. Local state is never lost when the process recycles or the VM reboots.

The primary use case for Durable Functions is simplifying complex, stateful coordination requirements in serverless applications. The following are some typical application patterns that can benefit from Durable Functions:

* Chaining
* Fan-out/fan-in
* Async HTTP APIs
* Monitoring
* Human interaction

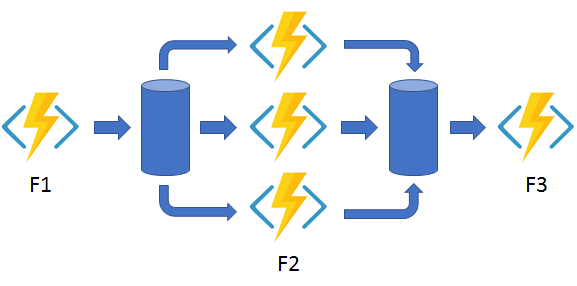
### Pattern #1: Function chaining

Function chaining refers to the pattern of executing a sequence of functions in a particular order. Often the output of one function needs to be applied to the input of another function.



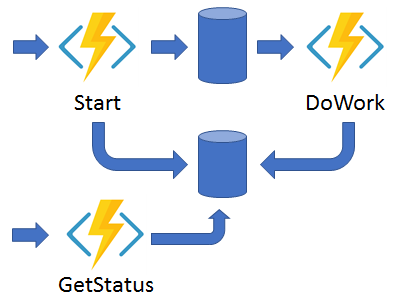
### Pattern #2: Fan-out/fan-in

Fan-out/fan-in refers to the pattern of executing multiple functions in parallel, and then waiting for all to finish. Often some aggregation work is done on results returned from the functions.



### Pattern #3: Async HTTP APIs

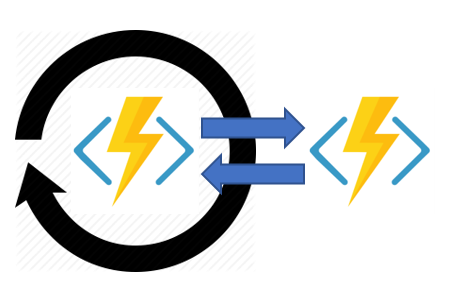
The third pattern is all about the problem of coordinating the state of long-running operations with external clients. A common way to implement this pattern is by having the long-running action triggered by an HTTP call, and then redirecting the client to a status endpoint that they can poll to learn when the operation completes.



### Pattern #4: Monitoring

The monitor pattern refers to a flexible recurring process in a workflow - for example, polling until certain conditions are met. A regular [timer trigger](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-timer) can address a simple scenario, such as a periodic cleanup job, but its interval is static and managing instance lifetimes becomes complex. Durable Functions enables flexible recurrence intervals, task lifetime management, and the ability to create multiple monitor processes from a single orchestration.

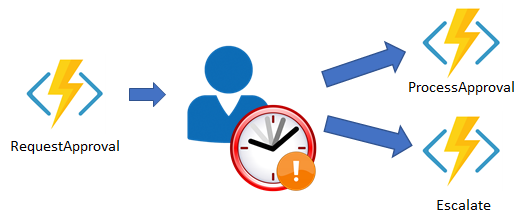
An example would be reversing the earlier async HTTP API scenario. Instead of exposing an endpoint for an external client to monitor a long-running operation, the long-running monitor consumes an external endpoint, waiting for some state change.



### Pattern #5: Human interaction

Many processes involve some kind of human interaction. The tricky thing about involving humans in an automated process is that people are not always as highly available and responsive as cloud services. Automated processes must allow for this, and they often do so by using timeouts and compensation logic.

One example of a business process that involves human interaction is an approval process. For example, approval from a manager might be required for an expense report that exceeds a certain amount. If the manager does not approve within 72 hours (maybe they went on vacation), an escalation process kicks in to get the approval from someone else (perhaps the manager's manager).



**Supported languages**

Durable Functions currently supports the following languages:

* **C#**: both precompiled class libraries and C# script.
* **F#**: precompiled class libraries and F# script. F# script is only supported for version 1.x of the Azure Functions runtime.
* **JavaScript**: supported only for version 2.x of the Azure Functions runtime. Requires version 1.7.0 of the Durable Functions extension, or a later version.