# Serverless applications

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

## Technology options

Azure includes four different technologies that you can use to build and implement workflows that integrate multiple systems. There is no requirement for you to use the same technology for different workflows.

### Design-first technologies

which include user interfaces in which you can draw out the workflow. Both technologies allow custom connectors.

* **Logic Apps**: (technical staff) Is a service within Azure that you can use to automate, orchestrate, and integrate disparate components of a distributed application. You can draw out complex workflows that model complex business processes. It provides hundreds of pre-built connectors that you can use to create your apps.
* **Microsoft Power Automate**: (non-technical staff) You can create workflows that integrate and orchestrate many different components by using the website or the Microsoft Power Automate mobile app.

### Code-first technologies

Use this approach when you need more control over the performance of your workflow or need to write custom code as part of the business process

* **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 loop) and *Triggered* (manually start or schedule). This technology has only two advantages over AZ functins: *Can be part of an App Service application* and *Provides close control of JobHost*
* **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. It has four trigger options: *HTTPTrigger*, *TimerTrigger*, *BlobTrigger* and *CosmosDBTrigger*. **IMPORTANT: it can run on a consumption plan!!!** 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.

## 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.

### Service plan

Function apps may use one of two types of service plans

* **Consumption service plan**: it provides automatic scaling and bills you when your functions are running. It has a configurable timeout as long as 10 minutes (by default has 5 minutes).
* **Azure App Service plan**: It allows to avoid timeout periods by having the function run continuously on a VM, so this is technically not a serverless plan. It may be a better choice if the functions are used continuously or if the functions require more processing power or execution time than the Consumption plan can provide.

### Storage account

The function app uses a 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

You must configure a function with exactly one trigger

* **Blob storage**: Start a function when a new or updated blob is detected.
* **Azure 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.

#### 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.

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

For example, to execute a function **every 30 seconds, between 10:00 PM and 11:00 PM, from Tuesdays to Wednesday and just in January and March** the CRON must be: **\*/30 \* 22 \* 1,3 2-5**

|  |  |  |
| --- | --- | --- |
| **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. |

#### HTTP trigger

An HTTP trigger is a trigger that executes a function when it receives an HTTP request. Azure Functions provides an Authorization level for its functions which are:

* **Function**: ("key" based) For this you can use a function key (specific to a function) or a host key (apply to all functions inside the function app).
* **Anonymous**: no authentication required.
* **Admin**: ("key" based) For this you must use a host key.

These kinds of functions can receive data from query strings or request body. Also, can returns data back to the caller.

#### 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. 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}***

* **samples-workitems**: The first part represents the blob container that the trigger monitors.
* **{name}**: The name represents a parameter in your Azure function that receives the name of the added file. Also after that you could set the filters, for example {fileName}.png

### Bindings

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.

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 ([Supported bindings](https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings#supported-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.

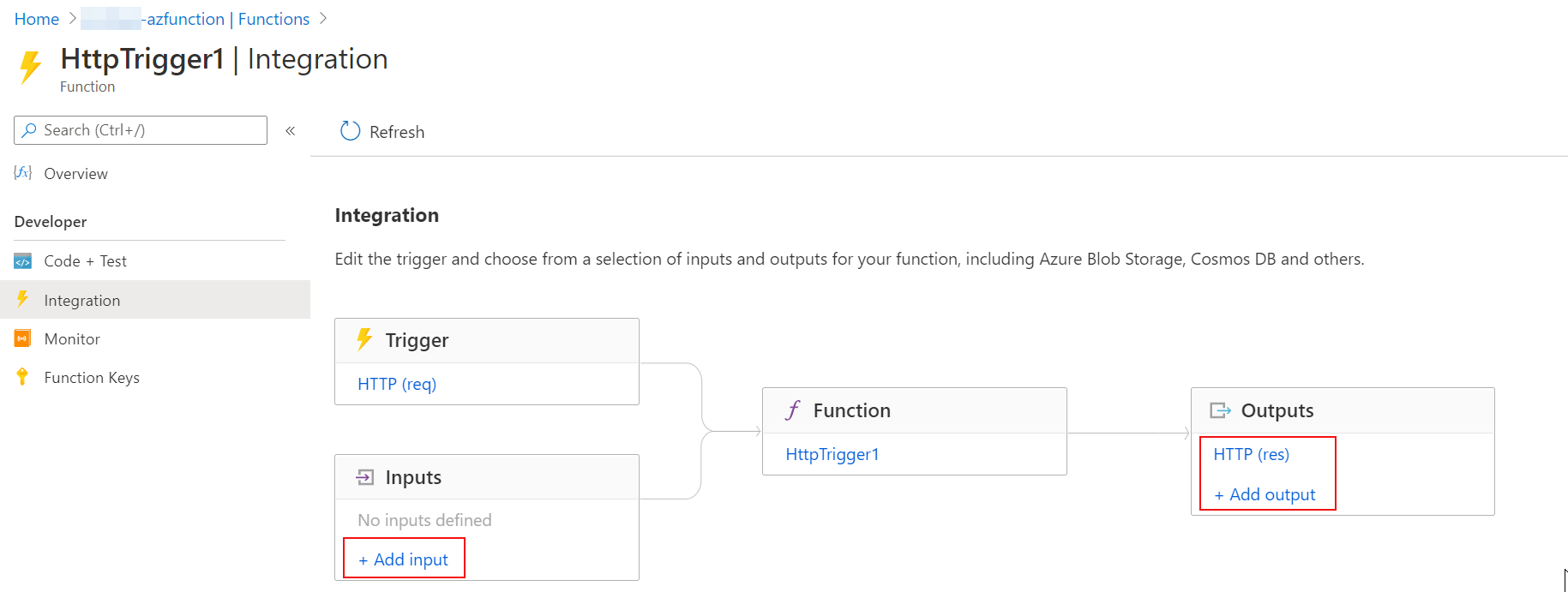
1. **Name** - Defines the function parameter through which you access the data. For example, in a queue input binding, this is the name of the function parameter that receives the queue message content.
2. **Type** - Identifies the type of binding, i.e., the type of data or service we want to interact 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.

#### Explore bindings

You can manage the bindings functions from the Integration section. There you can add/edit inputs and outputs bindings.



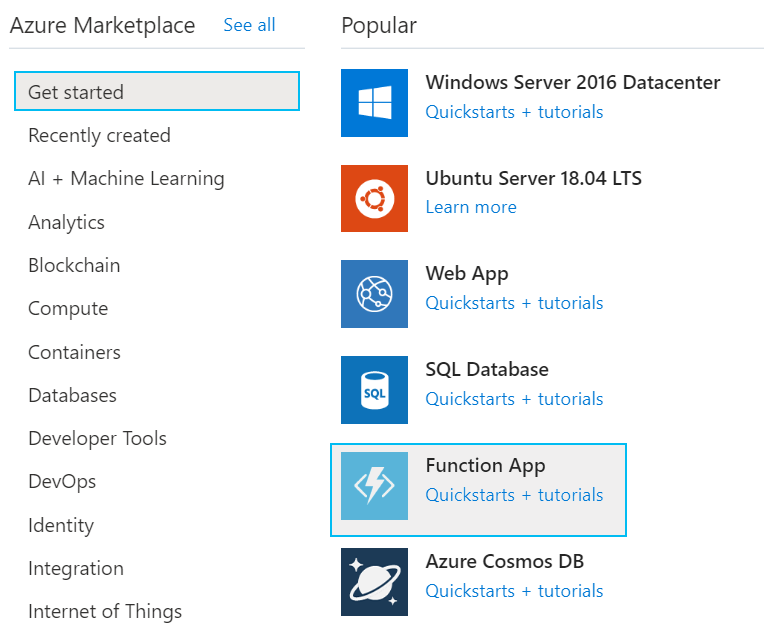
### Log window

You are able to add logging statements to your function for debugging in the Azure portal. The called methods for each language are passed a "logging" object, which may be used to log information to the log window located in a tabbed flyout menu located at the bottom of the code window.

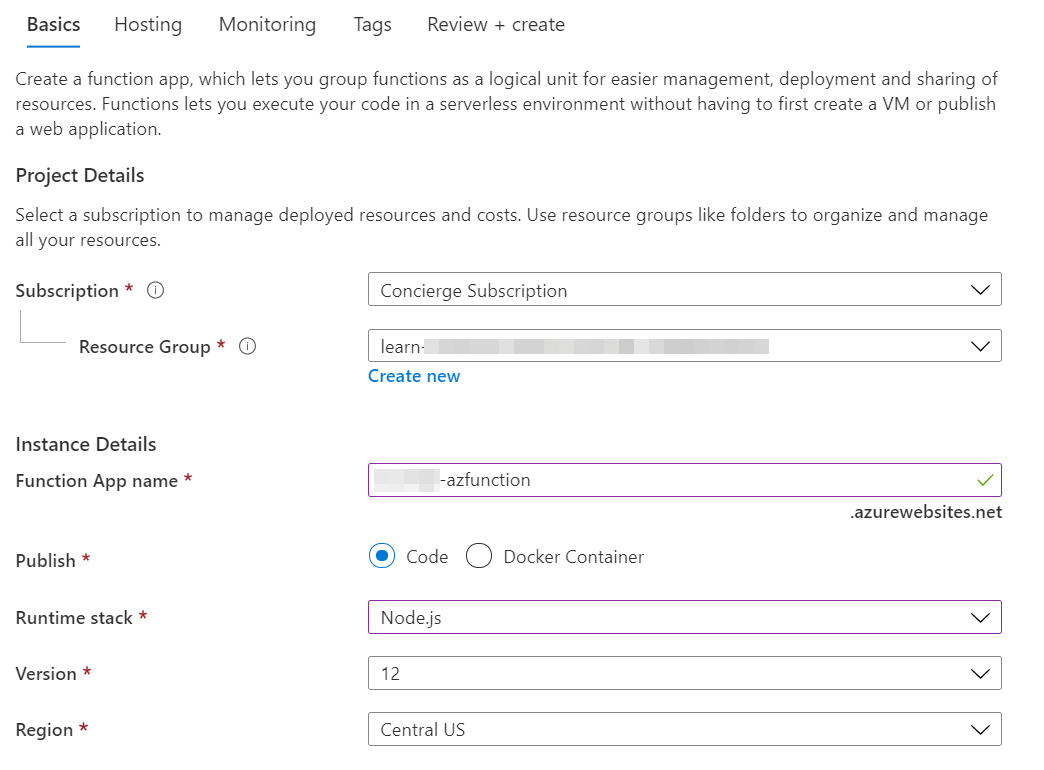
* **JavaScript**: context.log('Enter your logging statement here');
* **C#**: log.Info("Enter your logging statement here");
* **PowerShell**: Write-Host "Enter your logging statement here"

### Create a Function app

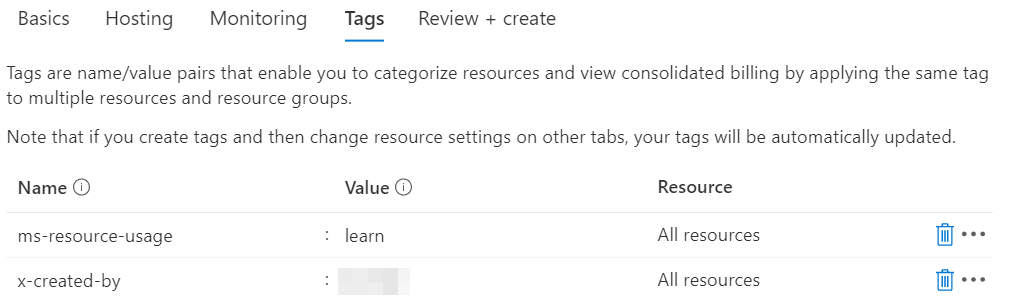
This is a summary to create e Function APP. For it you should go into the Resource-group and then choose Add (at the top-left of the page). After that choose Function App in the resources list.



Complete the Basics tab with a Function App name and the Runtime stack (.NET Core, Node.js, Python, Java or Powershell Core).

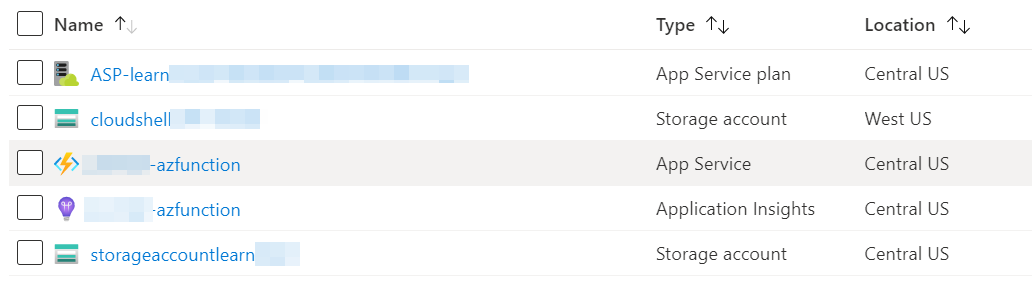


In the Tags section, you could add any tags as you need. This is not mandatory but is very recommended, because this way the resources are easily identifiable.

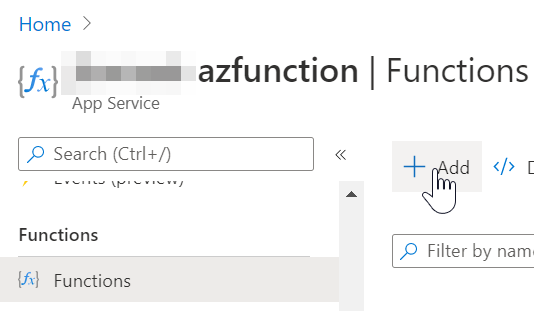


### Create an Azure Function from Portal

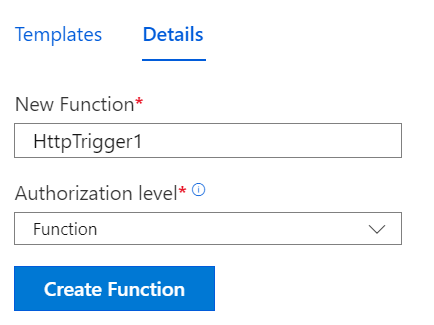
Once the function app is created, go into the App Service resource, and then go to Functions



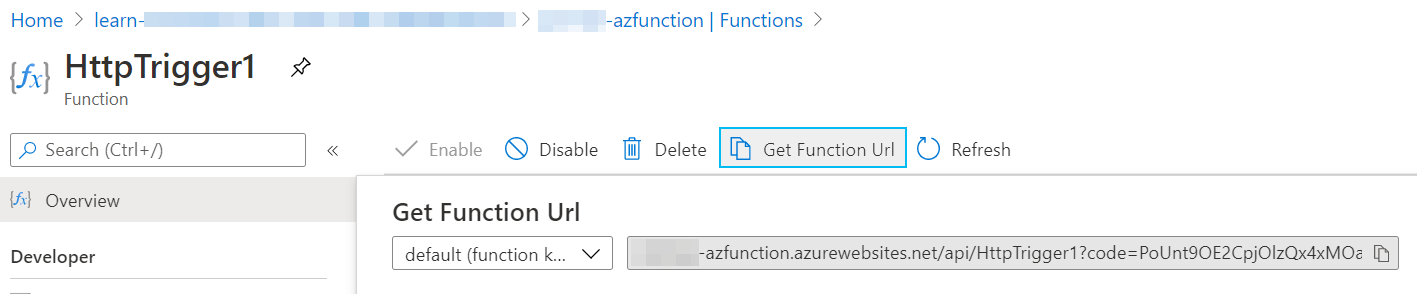
Once in there, in the top of the page you could see the button to add new Azure Functions



After that you should choose the Trigger (in this case will be HTTP trigger, but there are a lot of options), the name and the Authorization level.



If you select **function** as authorization level, the button **Get Function Url** will provide you an authorization code to call this endpoint



### Create an Azure Function from outside the portal

To create functions from outside the portal, we recommend the next tutorials from Microsoft Docs which detail clearly the steps

* From Visual Studio ([link](https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-your-first-function-visual-studio))
* From Visual Studio Code ([link](https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-function-vs-code?pivots=programming-language-typescript)) also it tutorial has the options to change the programming language

## 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.

### 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.
* **Orchestrator functions**:describe how actions are executed, and the order in which they are run.
* **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.

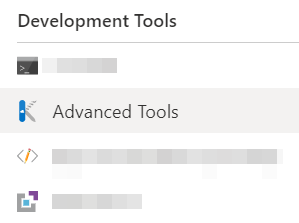
### Application patterns

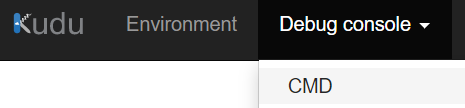
* **Function chaining** - In this pattern, the workflow executes a sequence of functions in a specified order. The output of one function is applied to the input of the next function in the sequence. The output of the final function is used to generate a result.
* **Fan out/fan in** - This pattern runs multiple functions in parallel and 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.

### Install npm packages

We will show how to add the durable-functions package, but the same process is for any package that you would like to install.

1. Go to Advanced Tools. It will open the Kudu console. Then go to **Debug conso**le > **CMD**





1. In the Explorer window, navigate to the **site** folder, and then select the **wwwroot** folder.
2. Click the + icon next to wwwroot, then select New file.
   1. Name the file **package.json**, then click the Edit icon to open the file in the editor.
   2. Enter the following JSON code, replacing the value for name with the globally unique name that you specified for you **Function App name** earlier.

{

"name": "example",

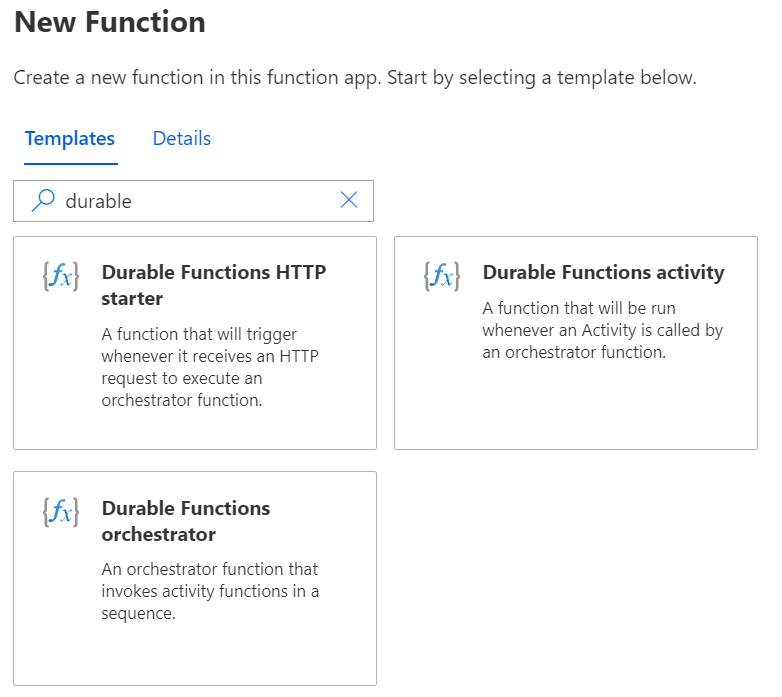
"version": "1.0.0"

}

1. In the command prompt window, verify that you're in the D:\home\site\wwwroot folder, then run the following command: *npm install durable-functions*
2. After the installation finish, go back to the azure portal, click the Overview tab, then click Restart.

### Create Durable functions

When you try to create a new Function into the Function App, you must first select the template. There are three different ones to durable functions.



The three of them are required to durable functions

1. HTTP starter: it runs when the user visits the web site hosting the function. It invokes an orchestration function.
2. Orchestrator: It function should call a series of activity functions that make the logic, and return the result of these activities.
3. Activity: These functions should have the logic an code to validate the data passed from the Orchestrator and return a result of it.

## Serverless architecture and microservices

When you build an application as a collection of microservices, you create many different small services. Each service has a defined domain of responsibility, and is developed, deployed, and scaled independently. This modular architecture results in an application that is easier to understand, improve, and test.

Composing an API using API Management has advantages that include:

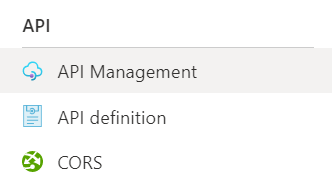
* Client apps are coupled to the API expressing business logic, not the underlying technical implementation with individual microservices. You can change the location and definition of the services without necessarily reconfiguring or updating the client apps.
* API Management acts as an intermediary. 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.

### Azure API Management

Azure API Management (APIM) 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.

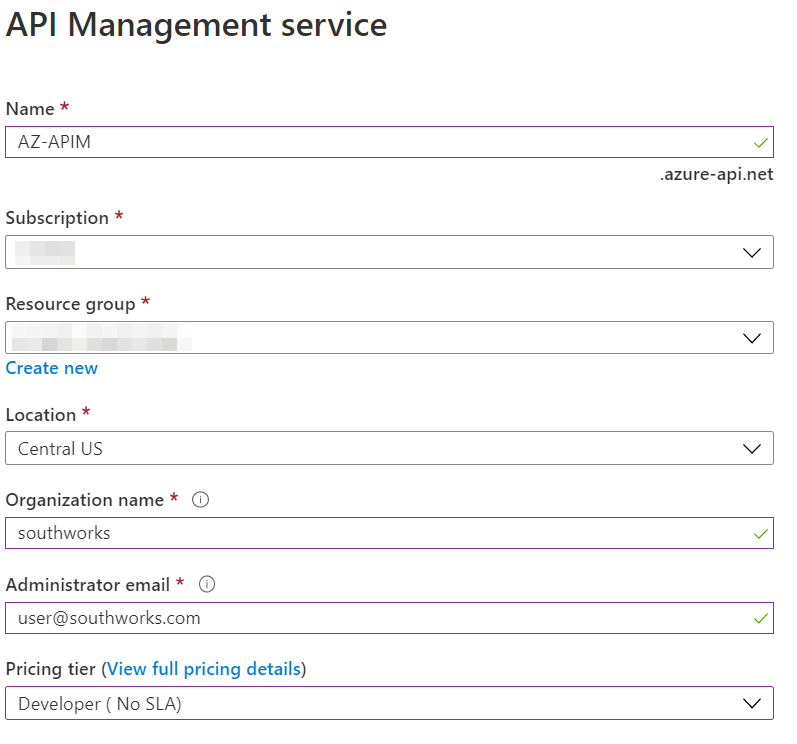
API Management also includes helpful tools - you can test each microservice and its operations to ensure that they behave in accordance with your requirements. You can also monitor the behavior and performance of deployed services.

You can associate Function APPs with an API Management from each one. For it go inside the Function App, then go to **API** > **API Management**. There you could create a new API Management or choose an existing one.



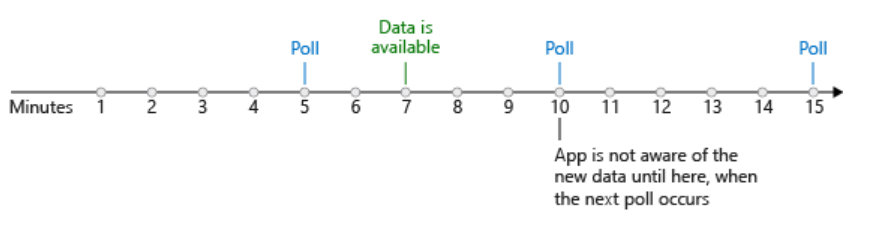
#### Create an API Management

Even if you choose to create an API Management from the Function App or from the New Resources window, you should complete the same form.



## Polling-based web app

In the timer-based polling prototype, the client application contacts the server whether or not changes exist to the underlying data. Once data is returned from the server the entire list of items is updated on the web page - again - regardless of any changes in the data. This polling mechanism is an inefficient solution. Polling forces you to make a choice between how much each call to the backend costs and how quickly you want your app to respond to new data. Delays also often exist between when new data becomes available and when it's detected by the app.



## Automatic updates in a web app

This design reduces traffic and makes a more efficient UI by only updating as data changes. The three technologies that we'll use to deliver this updated solution are **Azure Cosmos DB**, **Azure Functions**, and **SignalR**.

Azure Functions features a binding that runs code anytime data is updated in an Azure Cosmos DB change feed. Once a function is listening to the change feed, then you can work with a subset of your data that just represents data changes. When paired with a persistent connection to the client, the function can contact individual clients on-demand, which is the foundation for a real-time application architecture.

### SignalR and persistent connections

SignalR is an abstraction for a series of technologies that allows your app to enjoy two-way communication between the client and server. SignalR handles connection management automatically, and lets you broadcast messages to all connected clients simultaneously, or to specific clients.

A key benefit of the abstraction provided by SignalR is the way it supports "transport" fallbacks. A transport is method of communicating between the client and server. SignalR connections begin with a standard HTTP request. As the server evaluates the connection, the most appropriate communication method (transport) is selected. Transports are chosen depending on the APIs available on the client.

#### Manage client connections

The web client uses the SignalR client SDK to establish a connection to the server. The SDK retrieves the connection via a function named **negotiate** (by convention) to connect to the service. This function must have a **SignalR input binding**

#### Detect and broadcast database changes

First, you need to create a new function that listens for changes in the database. This function uses an **Azure Cosmos DB trigger** that connects to the change feed of the database. Next, append a **SignalR output binding** definition to the bindings array, this binding allows the function to broadcast changes to clients.

#### Establish connection and await messages

From the client side, the first action is to use the **SignalR SDK** to create a connection by calling **HubConnectionBuilder**. The result is a **SignalR connection** to the server. Then, ss the client receives messages from the server, it listens for messages via the **connection .on('updated',...** syntax.