The popular trend in today's technology driven world is ‘Cloud Computing’. Cloud computing can be referred to as the storing and accessing of data over the internet rather than your computer's hard drive. This means you don't access the data from either your computer's hard drive or over a dedicated computer network (home or office network). Cloud computing means data is stored at a remote place and is synchronized with other web information.

One prominent example of cloud computing is Office 365 which allows users to store, access, edit their MS Office documents online (in browser) without installing the actual program on their device.

Architecture of Cloud Computing

The architecture of cloud computing comprises of the following components −

* Front-end device
* Back-end platform
* Cloud-based delivery
* Network

**Front-end Devices** − These are basically the devices that are used by clients to access the data or program using the browser or special applications.

**Back-end Platform** − There are various computers, servers, virtual machines, etc. that combine to become a back-end platform.

Types of Cloud

The storage options on cloud is in 3 forms −

* Public
* Private
* Hybrid

**Public Cloud** − A service provider makes the clouds available to the general public which is termed as a public cloud. These clouds are accessed through internet by users. These are open to public and their infrastructure is owned and operated by service providers as in case of Google and Microsoft.

**Private Cloud** − These clouds are dedicated to a particular organization. That particular organization can use the cloud for storing the company's data, hosting business application, etc. The data stored on private cloud can't be shared with other organizations. The cloud is managed either by the organization itself or by the third party.

**Hybrid Cloud** − When two or more clouds are bound together to offer the advantage of both public and private clouds, they are termed as Hybrid Cloud. Organizations can use private clouds for sensitive application, while public clouds for non-sensitive applications. The hybrid clouds provide flexible, scalable and cost-effective solutions to the organizations.

## SPI

Next comes how cloud services are categorized. S stand for Software, P stands for Platform and I for Infrastructure in SPI. SaaS is Software as a service; PaaS is Platform as a service and IaaS is Infrastructure as a Service.

Following are the live examples of these models.

* **SAAS Model** − E-mail (Gmail, Yahoo, etc.)
* **PAAS Model** − Microsoft Azure
* **IAAS Model** − Amazon S3

**What is Azure?**

Azure is a complete cloud platform that can host your existing applications, streamline the development of new applications, and even enhance on-premises applications. Azure integrates the cloud services that you need to develop, test, deploy, and manage your applications—while taking advantage of the efficiencies of cloud computing.

By hosting your applications in Azure, you can start small and easily scale your application as your customer demand grows. Azure also offers the reliability that’s needed for high-availability applications, even including failover between different regions. The [Azure portal](https://portal.azure.com/) lets you easily manage all your Azure services. You can also manage your services programmatically by using service-specific APIs and templates.

The services provided by Microsoft Azure are PaaS and IaaS. Many programming languages and frameworks are supported by it.

## Azure as PaaS (Platform as a Service)

As the name suggests, a platform is provided to clients to develop and deploy software. The clients can focus on the application development rather than having to worry about hardware and infrastructure. It also takes care of most of the operating systems, servers and networking issues.

### Pros

* The overall cost is low as the resources are allocated on demand and servers are automatically updated.
* It is less vulnerable as servers are automatically updated and being checked for all known security issues. The whole process is not visible to developer and thus does not pose a risk of data breach.
* Since new versions of development tools are tested by the Azure team, it becomes easy for developers to move on to new tools. This also helps the developers to meet the customer’s demand by quickly adapting to new versions.

### Cons

* There are portability issues with using PaaS. There can be a different environment at Azure, thus the application might have to be adapted accordingly.

## Azure as IaaS (Infrastructure as a Service)

It is a managed compute service that gives complete control of the operating systems and the application platform stack to the application developers. It lets the user to access, manage and monitor the data centers by themselves.

## Design a relational database in a single database within Azure SQL Database using SSMS.

Follow these steps to create a blank single database.

select **Databases** in the Azure Marketplace section, and then click **SQL Database** in the **Featured** section. Click on Add button

Fill out the **SQL Database** form with the following information, as shown on the preceding image:

| **Setting** | **Suggested value** | **Description** |
| --- | --- | --- |
| **Database name** | yourDatabaseName | For valid database names, see [Database identifiers](https://docs.microsoft.com/en-us/sql/relational-databases/databases/database-identifiers). |
| **Subscription** | yourSubscription | For details about your subscriptions, see [Subscriptions](https://account.windowsazure.com/Subscriptions). |
| **Resource group** | yourResourceGroup | For valid resource group names, see [Naming rules and restrictions](https://docs.microsoft.com/en-us/azure/architecture/best-practices/naming-conventions). |
| **Select source** | Blank database | Specifies that a blank database should be created. |

Click **Server** to use an existing database server or create and configure a new database server. Either select an existing server or click **Create a new server** and fill out the **New server** form with the following information:

| **Setting** | **Suggested value** | **Description** |
| --- | --- | --- |
| **Server name** | Any globally unique name | For valid server names, see [Naming rules and restrictions](https://docs.microsoft.com/en-us/azure/architecture/best-practices/naming-conventions). |
| **Server admin login** | Any valid name | For valid login names, see [Database identifiers](https://docs.microsoft.com/en-us/sql/relational-databases/databases/database-identifiers). |
| **Password** | Any valid password | Your password must have at least eight characters and must use characters from three of the following categories: upper case characters, lower case characters, numbers, and non-alphanumeric characters. |
| **Location** | Any valid location | For information about regions, see [Azure Regions](https://azure.microsoft.com/regions/). |

Click **Pricing tier** to specify the service tier, the number of DTUs or vCores, and the amount of storage. You may explore the options for the number of DTUs/vCores and storage that is available to you for each service tier.

Enter a **Collation** for the blank database (for this tutorial, use the default value). For more information about collations

Now that you've completed the **SQL Database** form, click **Create** to provision the single database. This step may take a few minutes.

**Once the creation sql database has been completed you have set the firewall for the database and server.**

This firewall prevents external applications and tools from connecting to the server and any databases on the server unless a firewall rule allows their IP through the firewall. To enable external connectivity to your single database, you must first add an IP firewall rule for your IP address (or IP address range)

Select the newly created the database from resource tab.

Click on set server firewall.

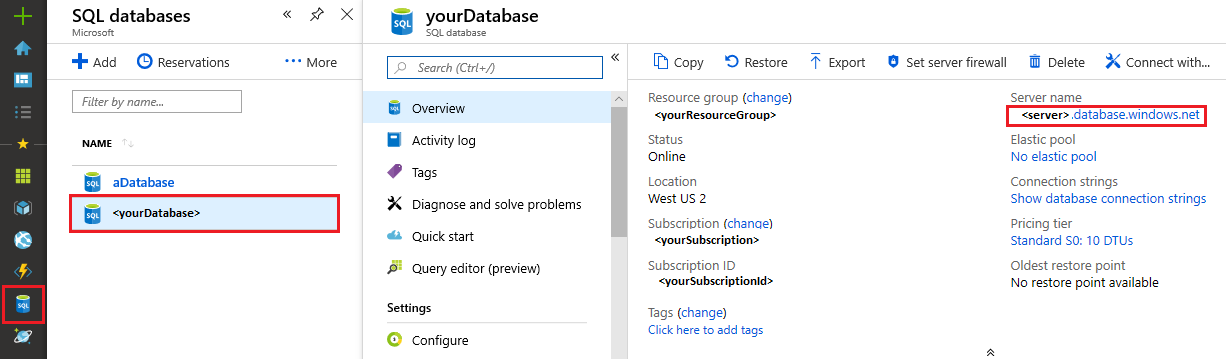
Click **Add client IP** on the toolbar to add your current IP address to a new IP firewall rule. An IP firewall rule can open port 1433 for a single IP address or a range of IP addresses.

Click Save. A server-level IP firewall rule is created for your current IP address opening port 1433 on the SQL Database server.

Click OK and then close the Firewall settings page.

**Connect to the database From SSMS**

Open the Sql server management studio on local server.



You will get the server name from newly created database overview shown above.

Enter the username and password of server those you have entered when creating the server.

Click on connect.

## Azure Function Or Serverless Function

## Azure Functions is a solution for easily running small pieces of code, or "functions," in the cloud. You can write just the code you need for the problem at hand, without worrying about a whole application or the infrastructure to run it. Functions can make development even more productive, and you can use your development language of choice, such as C#, F#, Node.js, Java, or PHP. Pay only for the time your code runs and trust Azure to scale as needed. Azure Functions lets you develop [serverless](https://azure.microsoft.com/solutions/serverless/) applications on Microsoft Azure.

## Features

Here are some key features of Functions:

* **Choice of language** - Write functions using your choice of C#, F#, or Javascript. See [Supported languages](https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages) for other options.
* **Pay-per-use pricing model** - Pay only for the time spent running your code. See the Consumption hosting plan option in the [pricing section](https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview#pricing).
* **Bring your own dependencies** - Functions supports NuGet and NPM, so you can use your favorite libraries.
* **Integrated security** - Protect HTTP-triggered functions with OAuth providers such as Azure Active Directory, Facebook, Google, Twitter, and Microsoft Account.
* **Simplified integration** - Easily leverage Azure services and software-as-a-service (SaaS) offerings. See the [integrations section](https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview#integrations) for some examples.
* **Flexible development** - Code your functions right in the portal or set up continuous integration and deploy your code through [GitHub](https://docs.microsoft.com/en-us/azure/app-service/scripts/cli-continuous-deployment-github), [Azure DevOps Services](https://docs.microsoft.com/en-us/azure/app-service/scripts/cli-continuous-deployment-vsts), and other [supported development tools](https://docs.microsoft.com/en-us/azure/app-service/deploy-local-git).
* **Open-source** - The Functions runtime is open-source and [available on GitHub](https://github.com/azure/azure-webjobs-sdk-script).

Functions is a great solution for processing data, integrating systems, working with the internet-of-things (IoT), and building simple APIs and microservices. Consider Functions for tasks like image or order processing, file maintenance, or for any tasks that you want to run on a schedule.

Azure Storage

Azure Storage is Microsoft's cloud storage solution for modern data storage scenarios. Azure Storage offers a massively scalable object store for data objects, a file system service for the cloud, a messaging store for reliable messaging, and a NoSQL store. Azure Storage is:

* **Durable and highly available.** Redundancy ensures that your data is safe in the event of transient hardware failures. You can also opt to replicate data across datacenters or geographical regions for additional protection from local catastrophe or natural disaster. Data replicated in this way remains highly available in the event of an unexpected outage.
* **Secure.** All data written to Azure Storage is encrypted by the service. Azure Storage provides you with fine-grained control over who has access to your data.
* **Scalable.** Azure Storage is designed to be massively scalable to meet the data storage and performance needs of today's applications.
* **Managed.** Microsoft Azure handles hardware maintenance, updates, and critical issues for you.
* **Accessible.** Data in Azure Storage is accessible from anywhere in the world over HTTP or HTTPS. Microsoft provides SDKs for Azure Storage in a variety of languages -- .NET, Java, Node.js, Python, PHP, Ruby, Go, and others -- as well as a mature REST API. Azure Storage supports scripting in Azure PowerShell or Azure CLI. And the Azure portal and Azure Storage Explorer offer easy visual solutions for working with your data.

## Azure Storage services

Azure Storage includes these data services:

* [Azure Blobs](https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction): A massively scalable object store for text and binary data.
* [Azure Files](https://docs.microsoft.com/en-us/azure/storage/files/storage-files-introduction): Managed file shares for cloud or on-premises deployments.
* [Azure Queues](https://docs.microsoft.com/en-us/azure/storage/queues/storage-queues-introduction): A messaging store for reliable messaging between application components.
* [Azure Tables](https://docs.microsoft.com/en-us/azure/storage/tables/table-storage-overview): A NoSQL store for schemaless storage of structured data.

**Blob storage**

Azure Blob storage is Microsoft's object storage solution for the cloud. Blob storage is optimized for storing massive amounts of unstructured data, such as text or binary data.

Blob storage is ideal for:

* Serving images or documents directly to a browser.
* Storing files for distributed access.
* Streaming video and audio.
* Storing data for backup and restore, disaster recovery, and archiving.
* Storing data for analysis by an on-premises or Azure-hosted service.

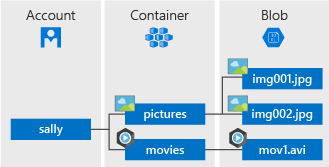
Objects in Blob storage can be accessed from anywhere in the world via HTTP or HTTPS. Users or client applications can access blobs via URLs, the [Azure Storage REST API](https://docs.microsoft.com/rest/api/storageservices/blob-service-rest-api), [Azure PowerShell](https://docs.microsoft.com/powershell/module/azure.storage), [Azure CLI](https://docs.microsoft.com/cli/azure/storage), or an Azure Storage client library. The storage client libraries are available for multiple languages, including [.NET](https://docs.microsoft.com/dotnet/api/overview/azure/storage/client), [Java](https://docs.microsoft.com/java/api/overview/azure/storage/client), [Node.js](https://azure.github.io/azure-storage-node), [Python](https://azure-storage.readthedocs.io/), [PHP](https://azure.github.io/azure-storage-php/), and [Ruby](https://azure.github.io/azure-storage-ruby).

**Blob storage resources**

Blob storage offers three types of resources:

* The **storage account**.
* A **container** in the storage account
* A **blob** in a container

The following diagram shows the relationship between these resources.



### Storage accounts

A storage account provides a unique namespace in Azure for your data. Every object that you store in Azure Storage has an address that includes your unique account name. The combination of the account name and the Azure Storage blob endpoint forms the base address for the objects in your storage account.

For example, if your storage account is named mystorageaccount, then the default endpoint for Blob storage is:

<http://mystorageaccount.blob.core.windows.net>

### Containers

A container organizes a set of blobs, similar to a directory in a file system. A storage account can include an unlimited number of containers, and a container can store an unlimited number of blobs.

### Blob Types

Azure Storage supports three types of blobs:

* **Block blobs** store text and binary data, up to about 4.7 TB. Block blobs are made up of blocks of data that can be managed individually.
* **Append blobs** are made up of blocks like block blobs, but are optimized for append operations. Append blobs are ideal for scenarios such as logging data from virtual machines.
* **Page blobs** store random access files up to 8 TB in size. Page blobs store the virtual hard drive (VHD) files serve as disks for Azure virtual machines. For more information about page blobs,

**Move data to Blob storage**

**AzCopy** is an easy-to-use command-line tool for Windows and Linux that copies data to and from Blob storage, across containers, or across storage accounts.

The **Azure Storage Data Movement library** is a .NET library for moving data between Azure Storage services. The AzCopy utility is built with the Data Movement library.

**Azure Data Factory** supports copying data to and from Blob storage by using the account key, shared access signature, service principal, or managed identities for Azure resources authentications.

**Blobfuse** is a virtual file system driver for Azure Blob storage. You can use blobfuse to access your existing block blob data in your Storage account through the Linux file system.

**Azure Data Box Disk** is a service for transferring on-premises data to Blob storage when large datasets or network constraints make uploading data over the wire unrealistic. You can use [Azure Data Box Disk](https://docs.microsoft.com/en-us/azure/databox/data-box-disk-overview) to request solid-state disks (SSDs) from Microsoft. You can then copy your data to those disks and ship them back to Microsoft to be uploaded into Blob storage.

The **Azure Import/Export service** provides a way to export large amounts of data from your storage account to hard drives that you provide and that Microsoft then ships back to you with your data

## Azure Files

[Azure Files](https://docs.microsoft.com/en-us/azure/storage/files/storage-files-introduction) enables you to set up highly available network file shares that can be accessed by using the standard Server Message Block (SMB) protocol. That means that multiple VMs can share the same files with both read and write access. You can also read the files using the REST interface or the storage client libraries.

One thing that distinguishes Azure Files from files on a corporate file share is that you can access the files from anywhere in the world using a URL that points to the file and includes a shared access signature (SAS) token. You can generate SAS tokens; they allow specific access to a private asset for a specific amount of time.

File shares can be used for many common scenarios:

* Many on-premises applications use file shares. This feature makes it easier to migrate those applications that share data to Azure. If you mount the file share to the same drive letter that the on-premises application uses, the part of your application that accesses the file share should work with minimal, if any, changes.
* Configuration files can be stored on a file share and accessed from multiple VMs. Tools and utilities used by multiple developers in a group can be stored on a file share, ensuring that everybody can find them, and that they use the same version.
* Diagnostic logs, metrics, and crash dumps are just three examples of data that can be written to a file share and processed or analyzed later.

At this time, Active Directory-based authentication and access control lists (ACLs) are not supported, but they will be at some time in the future. The storage account credentials are used to provide authentication for access to the file share. This means anybody with the share mounted will have full read/write access to the share.

## Queue storage

The Azure Queue service is used to store and retrieve messages. Queue messages can be up to 64 KB in size, and a queue can contain millions of messages. Queues are generally used to store lists of messages to be processed asynchronously.

For example, say you want your customers to be able to upload pictures, and you want to create thumbnails for each picture. You could have your customer wait for you to create the thumbnails while uploading the pictures. An alternative would be to use a queue. When the customer finishes his upload, write a message to the queue. Then have an Azure Function retrieve the message from the queue and create the thumbnails. Each of the parts of this processing can be scaled separately, giving you more control when tuning it for your usage.

## Table storage

Azure Table storage is now part of Azure Cosmos DB. To see Azure Table storage documentation, see the [Azure Table Storage Overview](https://docs.microsoft.com/en-us/azure/storage/tables/table-storage-overview). In addition to the existing Azure Table storage service, there is a new Azure Cosmos DB Table API offering that provides throughput-optimized tables, global distribution, and automatic secondary indexes.

## Disk storage

Azure Storage also includes managed and unmanaged disk capabilities used by virtual machines

**Types of storage accounts**

Azure Storage offers several types of storage accounts. Each type supports different features and has its own pricing model. Consider these differences before you create a storage account to determine the type of account that is best for your applications. The types of storage accounts are:

* **General-purpose v2 accounts**: Basic storage account type for blobs, files, queues, and tables. Recommended for most scenarios using Azure Storage.
* **General-purpose v1 accounts**: Legacy account type for blobs, files, queues, and tables. Use general-purpose v2 accounts instead when possible.
* **Block blob storage accounts**: Blob-only storage accounts with premium performance characteristics. Recommended for scenarios with high transactions rates, using smaller objects, or requiring consistently low storage latency.
* **FileStorage (preview) storage accounts**: Files-only storage accounts with premium performance characteristics. Recommended for enterprise or high performance scale applications.
* **Blob storage accounts**: Blob-only storage accounts. Use general-purpose v2 accounts instead when possible.

**Securing access to storage accounts**

Every request to Azure Storage must be authorized. Azure Storage supports the following authorization methods:

* **Azure Active Directory (Azure AD) integration for blob and queue data.** Azure Storage supports authentication and authorization with Azure AD credentials for the Blob and Queue services via role-based access control (RBAC). Authorizing requests with Azure AD is recommended for superior security and ease of use.
* **Azure AD authorization over SMB for Azure Files (preview).** Azure Files supports identity-based authorization over SMB (Server Message Block) through Azure Active Directory Domain Services. Your domain-joined Windows virtual machines (VMs) can access Azure file shares using Azure AD credentials. For more information, see [Overview of Azure Active Directory authorization over SMB for Azure Files (preview)](https://docs.microsoft.com/en-us/azure/storage/files/storage-files-active-directory-overview).
* **Authorization with Shared Key.** The Azure Storage Blob, Queue, and Table services and Azure Files support authorization with Shared Key.A client using Shared Key authorization passes a header with every request that is signed using the storage account access key.
* **Authorization using shared access signatures (SAS).** A shared access signature (SAS) is a string containing a security token that can be appended to the URI for a storage resource. The security token encapsulates constraints such as permissions and the interval of access.
* **Anonymous access to containers and blobs.** A container and its blobs may be publicly available. When you specify that a container or blob is public, anyone can read it anonymously; no authentication is required.

## Encryption

There are two basic kinds of encryption available for the Storage services. For more information about security and encryption, see the [Azure Storage security guide](https://docs.microsoft.com/en-us/azure/storage/common/storage-security-guide).

### Encryption at rest

Azure Storage Service Encryption (SSE) at rest helps you protect and safeguard your data to meet your organizational security and compliance commitments. With this feature, Azure Storage automatically encrypts your data prior to persisting to storage and decrypts prior to retrieval. The encryption, decryption, and key management are totally transparent to users.

SSE automatically encrypts data in all performance tiers (Standard and Premium), all deployment models (Azure Resource Manager and Classic), and all of the Azure Storage services (Blob, Queue, Table, and File). SSE does not affect Azure Storage performance.

For more information about SSE encryption at rest, see [Azure Storage Service Encryption for Data at Rest](https://docs.microsoft.com/en-us/azure/storage/common/storage-service-encryption).

### Client-side encryption

The storage client libraries have methods you can call to programmatically encrypt data before sending it across the wire from the client to Azure. It is stored encrypted, which means it also is encrypted at rest. When reading the data back, you decrypt the information after receiving it.

## Redundancy

In order to ensure that your data is durable, Azure Storage replicates multiple copies of your data. When you set up your storage account, you select a redundancy option.

Replication options for a storage account include:

* [Locally-redundant storage (LRS)](https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy-lrs): A simple, low-cost replication strategy. Data is replicated within a single storage scale unit.
* [Zone-redundant storage (ZRS)](https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy-zrs): Replication for high availability and durability. Data is replicated synchronously across three availability zones.
* [Geo-redundant storage (GRS)](https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy-grs): Cross-regional replication to protect against region-wide unavailability.
* [Read-access geo-redundant storage (RA-GRS)](https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy-grs#read-access-geo-redundant-storage): Cross-regional replication with read access to the replica.

# Deciding when to use Azure Blobs, Azure Files, or Azure Disks

* ‎11‎/‎28‎/‎2018
* 3 minutes to read
* Contributors



* + [all](https://github.com/Microsoft/azure-docs/blob/master/articles/storage/common/storage-decide-blobs-files-disks.md)

Microsoft Azure provides several features in Azure Storage for storing and accessing your data in the cloud. This article covers Azure Files, Blobs, and Disks, and is designed to help you choose between these features.

## Scenarios

The following table compares Files, Blobs, and Disks, and shows example scenarios appropriate for each.

| **Feature** | **Description** | **When to use** |
| --- | --- | --- |
| **Azure Files** | Provides an SMB interface, client libraries, and a [REST interface](https://docs.microsoft.com/en-us/rest/api/storageservices/file-service-rest-api) that allows access from anywhere to stored files. | You want to "lift and shift" an application to the cloud which already uses the native file system APIs to share data between it and other applications running in Azure.  You want to store development and debugging tools that need to be accessed from many virtual machines. |
| **Azure Blobs** | Provides client libraries and a [REST interface](https://docs.microsoft.com/en-us/rest/api/storageservices/blob-service-rest-api) that allows unstructured data to be stored and accessed at a massive scale in block blobs.  Also supports [Azure Data Lake Storage Gen2](https://docs.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-introduction) for enterprise big data analytics solutions. | You want your application to support streaming and random access scenarios.  You want to be able to access application data from anywhere.  You want to build an enterprise data lake on Azure and perform big data analytics. |
| **Azure Disks** | Provides client libraries and a [REST interface](https://docs.microsoft.com/en-us/rest/api/compute/manageddisks/disks/disks-rest-api) that allows data to be persistently stored and accessed from an attached virtual hard disk. | You want to lift and shift applications that use native file system APIs to read and write data to persistent disks.  You want to store data that is not required to be accessed from outside the virtual machine to which the disk is attached. |

## Comparison: Files and Blobs

The following table compares Azure Files with Azure Blobs.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Azure Blobs** | **Azure Files** |
| Durability options | LRS, ZRS, GRS, RA-GRS | LRS, ZRS, GRS |
| Accessibility | REST APIs | REST APIs  SMB 2.1 and SMB 3.0 (standard file system APIs) |
| Connectivity | REST APIs -- Worldwide | REST APIs - Worldwide  SMB 2.1 -- Within region  SMB 3.0 -- Worldwide |
| Endpoints | http://myaccount.blob.core.windows.net/mycontainer/myblob | \\myaccount.file.core.windows.net\myshare\myfile.txt  http://myaccount.file.core.windows.net/myshare/myfile.txt |
| Directories | Flat namespace | True directory objects |
| Case sensitivity of names | Case sensitive | Case insensitive, but case preserving |
| Capacity | Up to 2 PiB Account Limit | 5 TiB file shares |
| Throughput | Up to 60 MiB/s per block blob | Up to 60 MiB/s per share |
| Object Size | Up to about 4.75 TiB per block blob | Up to 1 TiB per file |
| Billed capacity | Based on bytes written | Based on file size |
| Client libraries | Multiple languages | Multiple languages |

## Comparison: Files and Disks

Azure Files complement Azure Disks. A disk can only be attached to one Azure Virtual Machine at a time. Disks are fixed-format VHDs stored as page blobs in Azure Storage, and are used by the virtual machine to store durable data. File shares in Azure Files can be accessed in the same way as the local disk is accessed (by using native file system APIs), and can be shared across many virtual machines.

The following table compares Azure Files with Azure Disks.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Azure Disks** | **Azure Files** |
| Scope | Exclusive to a single virtual machine | Shared access across multiple virtual machines |
| Snapshots and Copy | Yes | Yes |
| Configuration | Connected at startup of the virtual machine | Connected after the virtual machine has started |
| Authentication | Built-in | Set up with net use |
| Access using REST | Files within the VHD cannot be accessed | Files stored in a share can be accessed |
| Max Size | 4 TiB disk | 5 TiB File Share and 1 TiB file within share |
| Max IOps | 500 IOps | 1000 IOps |
| Throughput | Up to 60 MiB/s per Disk | Target is 60 MiB/s per File Share (can get higher for higher IO sizes) |

## App Service

A WebJob is a program running in the background of an App Service. It runs in the same context as your web app at no additional cost. Maybe you need to do some hourly task or do some cleanup task every night at 1 AM. Azure Application Insights uses a WebJob to report your app's statistics.

WebJob can be scheduled, like hourly or daily, but they can also be triggered. A trigger could be a file upload or a new message on a queue.

[*WebJobs*](https://docs.microsoft.com/en-us/azure/app-service-web/websites-webjobs-resources) *is a feature of Azure App Service that enables you to run a program or script in the same context as a web app, API app, or mobile app. The purpose of the* [*WebJobs SDK*](https://docs.microsoft.com/en-us/azure/app-service-web/websites-webjobs-resources) *is to simplify the code you write for common tasks that a WebJob can perform, such as image processing, queue processing, RSS aggregation, file maintenance, and sending emails.*

### WebJobs vs. Functions

I've often found comparisons between WebJobs and Azure Functions. In a way, Functions are the successors to WebJobs. Functions are (usually) small pieces of code that run in Azure and are, just like WebJobs, triggered at a certain event, including an HTTP trigger.

Functions are often a great alternative to WebJobs, but if you already have a web app it could make sense to use a WebJob instead. Especially if you want to share code and/or settings between the WebJob and your web app as they run in the same context, which also makes deployment quite easy.

**Creating a WebJob**

Azure WebJobs can be created using the following scripts:

1. .cmd, .bat, .exe (using windows cmd)
2. .ps1 (using powershell)
3. .sh (using bash)
4. .php (using php)
5. .py (using python)
6. .js (using node)

there's a WebJob template for the full .NET Framework. I recommend you check it out. Start by creating an ASP.NET Web Application and then add a new WebJob. If you try to create the WebJob right away you'll get an error saying that the project needs to be saved first (although it does create the WebJob).

We're here for the .NET Core version of a WebJob though. So start by creating an ASP.NET Core Web Application and then add a new .NET Core Console App project to your solution.

The first thing we need to do to is install the Microsoft.Azure.WebJobs package from NuGet. We should also install Microsoft.Azure.WebJobs.Extensions. Here's the catch though, the latest stable versions of these libraries have dependencies on the full .NET Framework so we're going to need version 3.0.0-beta5 (at the time of this writing), which is fully compatible with .NET Core.

Other NuGet packages we'll need are Microsoft.Extensions.Options.ConfigurationExtensions (which also gives us the Microsoft.Extensions.Options package, which we also need), Microsoft.Extensions.DependencyInjection and Microsoft.Extensions.Logging.Console. Be sure to install version 2.1.0 of these packages because there seems to be a bug in .NET Core 2.1 that prevents you from using packages with patch versions, like 2.1.1.

## Join the Program

The next thing we need to do is change our Program.cs file. If you've created a WebJob using the .NET Framework template you can simply copy and paste the Program.cs file that was generated there (although you might want to change the namespace).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | using Microsoft.Azure.WebJobs;    namespace NetCoreWebJob.WebJob  {      // To learn more about Microsoft Azure WebJobs SDK, please see <https://go.microsoft.com/fwlink/?LinkID=320976>      internal class Program      {          // Please set the following connection strings in app.config for this WebJob to run:          // AzureWebJobsDashboard and AzureWebJobsStorage          private static void Main()          {              var config = new JobHostConfiguration();              if (config.IsDevelopment)              {                  config.UseDevelopmentSettings();              }              var host = new JobHost(config);              // The following code ensures that the WebJob will be running continuously              host.RunAndBlock();          }      }  } |

### Adding Configuration and DI

So I promised you'd get all the .NET Core goodies like logging and DI. By default, a Console App doesn't have any of that, but you can add it yourself.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | private static void Main()  {      IServiceCollection services = new ServiceCollection();      ConfigureServices(services);        // ...  }    private static IConfiguration Configuration { get; set; }    private static void ConfigureServices(IServiceCollection services)  {      var environment = Environment.GetEnvironmentVariable("ASPNETCORE\_ENVIRONMENT");        Configuration = new ConfigurationBuilder()          .SetBasePath(Directory.GetCurrentDirectory())          .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)          .AddJsonFile($"appsettings.{environment}.json", optional: true, reloadOnChange: true)          .AddEnvironmentVariables()          .Build();        services.AddSingleton(Configuration);      services.AddTransient<Functions, Functions>();      services.AddLogging(builder => builder.AddConsole());  } |

Next, create an appsettings.json file and set the "Copy to Output Directory" property to "Copy always". The appsettings.json file should have two connection strings as mentioned in the Program.cs template file. These are the Storage Account connection strings we created earlier.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | {    "Logging": {      "LogLevel": {        "Default": "Warning"      }    },    "ConnectionStrings": {      "AzureWebJobsDashboard": "[your Storage Account connection string]",      "AzureWebJobsStorage": "[your Storage Account connection string]"    }  } |

The next thing we need is a custom IJobActivator that can be used to inject dependencies into our classes. It needs to be set on the JobHostConfiguration in the Program class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | using Microsoft.Azure.WebJobs.Host;  using Microsoft.Extensions.DependencyInjection;  using System;    namespace NetCoreWebJob.WebJob  {      public class JobActivator : IJobActivator      {          private readonly IServiceProvider services;            public JobActivator(IServiceProvider services)          {              this.services = services;          }            public T CreateInstance<T>()          {              return services.GetService<T>();          }      }  } |

|  |  |
| --- | --- |
| 1  2 | var config = new JobHostConfiguration();  config.JobActivator = new JobActivator(services.BuildServiceProvider()); |

### Adding a Trigger

After that, create a class and name it Functions (just like in the WebJob template). The Functions class will have the actual code for our WebJob.

Of course, we'll need to add a trigger. This is different than the full .NET Framework. After all, the template uses a static method, which makes DI impossible. Speaking of DI, notice that we've also added the Functions class itself to the DI container.

For simplicity, we'll use a TimerTrigger, which is triggered with a so-called [CRON expression](https://en.wikipedia.org/wiki/Cron). This simply means it's triggered at a certain minute, hour, day, etc. In this example, it triggers every minute.

We'll also need to configure timers on the JobHostConfiguration.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | using Microsoft.Azure.WebJobs;  using Microsoft.Extensions.Logging;  using System;    namespace NetCoreWebJob.WebJob  {      public class Functions      {          private readonly ILogger<Functions> logger;            public Functions(ILogger<Functions> logger)          {              this.logger = logger;          }            public void ProcessQueueMessage([TimerTrigger("\* \* \* \* \*")]TimerInfo timerInfo)          {              logger.LogInformation(DateTime.Now.ToString());          }      }  } |

|  |  |
| --- | --- |
| 1  2  3 | var config = new JobHostConfiguration();  config.JobActivator = new JobActivator(services.BuildServiceProvider());  config.UseTimers(); |

### Running the example

If you did everything correctly, or if you're running my code from GitHub, you should now be able to run the Console App. If you break on exceptions or if you're watching the Output window you may notice a lot of StorageExceptions. Don't worry about them and ignore them. This is a bug in the WebJobs library and will not affect your program. It may take a minute for your trigger to go off, so have a little patience.

If you head over to your Azure Storage Account you should see two Blob Containers, "azure-jobs-host-output" and "azure-webjobs-hosts". There's quite a lot going on here, but you can just ignore it. I've found that my WebJob triggers wouldn't go off for some reason, deleting the Blob Containers usually helped. Apparently, there's some state stored in there which isn't always disposed of properly when (re-)adding and removing WebJobs.

## Deploying to Azure

The next thing we want to do is deploy our WebJob to Azure. In order for a WebJob to run it needs some executable script that it can use to get going. Many file types are supported, but for us Windows users it makes sense to use an exe, cmd, bat or PowerShell file.

A Console App used to be an exe file, but in .NET Core, it produces a regular DLL file that we need to start manually. So, create a file and name it "run.cmd" and make sure that it's encoded in UTF-8 no BOM (you can check this using something like Notepad++). It just needs a single line of code, which is "dotnet NetCoreWebJob.WebJob.dll". This runs your Console App. Make sure you set the "Copy to Output Directory" of the file to "Copy always".

One last thing, for some reason Azure WebJobs needs all the dependencies of a WebJob, which means all .NET Core packages we used to build it. You can do this by editing the csproj file and adding "<CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>" to the first <PropertyGroup> (underneath "<TargetFramework>").

Before we can deploy our WebJob we need to deploy our web app. Right-click the ASP.NET project and click "Publish...". Simply follow the wizard and Visual Studio will deploy your app for you. You can create a new web app or select an existing one. This step isn't strictly necessary as you can host stand-alone WebJobs, but this should be familiar and it gives you an App Service we can use for our WebJob.

### Deploy using Visual Studio

Deploying WebJobs using Visual Studio should be easy as pie. In fact, you probably already know how to do this (don't do it yet though). Right-click your WebJob project and click "Publish...". The following wizard looks a whole lot like the publication of a web app, which we just did. You can pick "Select existing" and pick the Azure web app we just created.

Unfortunately, Microsoft messed up this feature in the worst way possible. Visual Studio will deploy the WebJob with the same name as the project, which is "NetCoreWebJob.WebJob", except the dot is an illegal character in a WebJob name! This messed up my project so bad I had to manually edit it to make my solution working again. Nice one, Microsoft!

So here's what you do. At the start of the wizard, where you pick either a new or existing App Service, click the arrow next to "Publish immediately" and pick "Create Profile" instead. Now you can first change the name of your WebJob in the settings and deploy after that. Make sure you don't select "Remove additional files at destination"  or you'll remove your web app.

Now, browse to the Azure Portal and look up your web app. You'll find "WebJobs" in the menu. You'll see your WebJob, but it's not doing anything. You need to manually run it by selecting it and clicking "Run". The status should update to "Running". You can now check out the logs to see that it actually works. You may see an error about the connection strings, but you can ignore those. If you toggle the output you'll still see a log is written to the console which lets you know it works! If you don't see a log right away try waiting a minute or two and don't forget to manually refresh the output.

## WebJobs SDK versions

These are the key differences between version 3.x and version 2.x of the WebJobs SDK:

* Version 3.x adds support for .NET Core.
* In version 3.x, you need to explicitly install the Storage binding extension required by the WebJobs SDK. In version 2.x, the Storage bindings were included in the SDK.
* Visual Studio tooling for .NET Core (3.x) projects differs from tooling for .NET Framework (2.x) projects. To learn more, see [Develop and deploy WebJobs using Visual Studio - Azure App Service](https://docs.microsoft.com/en-us/azure/app-service/webjobs-dotnet-deploy-vs).

When possible, examples are provided for both version 3.x and version 2.x.

## WebJobs host

The host is a runtime container for functions. It listens for triggers and calls functions. In version 3.x, the host is an implementation of IHost. In version 2.x, you use the JobHost object. You create a host instance in your code and write code to customize its behavior.

This is a key difference between using the WebJobs SDK directly and using it indirectly through Azure Functions. In Azure Functions, the service controls the host, and you can't customize the host by writing code. Azure Functions lets you customize host behavior through settings in the host.json file. Those settings are strings, not code, and this limits the kinds of customizations you can do.

### Host connection strings

The WebJobs SDK looks for Azure Storage and Azure Service Bus connection strings in the local.settings.json file when you run locally, or in the environment of the WebJob when you run in Azure. By default, a storage connection string setting named AzureWebJobsStorage is required.

**Version 2.x** of the SDK lets you use your own names for these connection strings or store them elsewhere. You can set names in code using the [JobHostConfiguration](https://github.com/Azure/azure-webjobs-sdk/blob/v2.x/src/Microsoft.Azure.WebJobs.Host/JobHostConfiguration.cs), as shown here:

static void Main(string[] args)

{

var \_storageConn = ConfigurationManager

.ConnectionStrings["MyStorageConnection"].ConnectionString;

JobHostConfiguration config = new JobHostConfiguration();

config.StorageConnectionString = \_storageConn;

JobHost host = new JobHost(config);

host.RunAndBlock();

}

#### Version 3.x

Version 3.x uses the standard ASP.NET Core APIs. Call the [UseEnvironment](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.hosting.hostinghostbuilderextensions.useenvironment) method on the [HostBuilder](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.hosting.hostbuilder) instance. Pass a string named development, as in this example:

static void Main()

{

var builder = new HostBuilder();

builder.UseEnvironment("development");

builder.ConfigureWebJobs(b =>

{

b.AddAzureStorageCoreServices();

});

var host = builder.Build();

using (host)

{

host.Run();

}

}

**Managing concurrent connections (version 2.x)**

In version 3.x, the connection limit defaults to infinite connections. If for some reason you need to change this limit, you can use the [MaxConnectionsPerServer](https://docs.microsoft.com/en-us/dotnet/api/system.net.http.winhttphandler.maxconnectionsperserver) property of the [WinHttpHandler](https://docs.microsoft.com/en-us/dotnet/api/system.net.http.winhttphandler) class.

In version 2.x, you control the number of concurrent connections to a host by using the [ServicePointManager.DefaultConnectionLimit](https://docs.microsoft.com/en-us/dotnet/api/system.net.servicepointmanager.defaultconnectionlimit#System_Net_ServicePointManager_DefaultConnectionLimit) API. In 2.x, you should increase this value from the default of 2 before starting your WebJobs host.

All outgoing HTTP requests that you make from a function by using HttpClient flow through ServicePointManager. After you reach the value set in DefaultConnectionLimit, ServicePointManager starts queueing requests before sending them. Suppose your DefaultConnectionLimit is set to 2 and your code makes 1,000 HTTP requests. Initially, only two requests are allowed through to the OS. The other 998 are queued until there’s room for them. That means your HttpClient might time out because it appears to have made the request, but the request was never sent by the OS to the destination server. So you might see behavior that doesn't seem to make sense: your local HttpClient is taking 10 seconds to complete a request, but your service is returning every request in 200 ms.

The default value for ASP.NET applications is Int32.MaxValue, and that's likely to work well for WebJobs running in a Basic or higher App Service Plan. WebJobs typically need the Always On setting, and that's supported only by Basic and higher App Service Plans.

If your WebJob is running in a Free or Shared App Service Plan, your application is restricted by the App Service sandbox, which currently has a [connection limit of 300](https://github.com/projectkudu/kudu/wiki/Azure-Web-App-sandbox#per-sandbox-per-appper-site-numerical-limits). With an unbound connection limit in ServicePointManager, it's more likely that the sandbox connection threshold will be reached and the site will shut down. In that case, setting DefaultConnectionLimit to something lower, like 50 or 100, can prevent this from happening and still allow for sufficient throughput.

The setting must be configured before any HTTP requests are made. For this reason, the WebJobs host shouldn't adjust the setting automatically. There could be HTTP requests that occur before the host starts, which could lead to unexpected behavior. The best approach is to set the value immediately in your Main method before initializing JobHost, as shown here:

static void Main(string[] args)

{

// Set this immediately so that it's used by all requests.

ServicePointManager.DefaultConnectionLimit = Int32.MaxValue;

var host = new JobHost();

host.RunAndBlock();

}

## Triggers

Functions must be public methods and must have one trigger attribute or the [NoAutomaticTrigger](https://docs.microsoft.com/en-us/azure/app-service/webjobs-sdk-how-to#manual-triggers) attribute.

### Automatic triggers

Automatic triggers call a function in response to an event. Consider this example of a function that's triggered by a message added to Azure Queue storage. It responds by reading a blob from Azure Blob storage:

public static void Run(

[QueueTrigger("myqueue-items")] string myQueueItem,

[Blob("samples-workitems/{myQueueItem}", FileAccess.Read)] Stream myBlob,

ILogger log)

{

log.LogInformation($"BlobInput processed blob\n Name:{myQueueItem} \n Size: {myBlob.Length} bytes");

}

The QueueTrigger attribute tells the runtime to call the function whenever a queue message appears in the myqueue-items queue. The Blob attribute tells the runtime to use the queue message to read a blob in the sample-workitems container. The content of the queue message, passed in to the function in the myQueueItem parameter, is the name of the blob.

### Manual triggers

To trigger a function manually, use the NoAutomaticTrigger attribute, as shown here:

[NoAutomaticTrigger]

public static void CreateQueueMessage(

ILogger logger,

string value,

[Queue("outputqueue")] out string message)

{

message = value;

logger.LogInformation("Creating queue message: ", message);

}

The process for manually triggering the function depends on the SDK version.

#### Version 3.x

static async Task Main(string[] args)

{

var builder = new HostBuilder();

builder.ConfigureWebJobs(b =>

{

b.AddAzureStorageCoreServices();

b.AddAzureStorage();

});

var host = builder.Build();

using (host)

{

var jobHost = host.Services.GetService(typeof(IJobHost)) as JobHost;

var inputs = new Dictionary<string, object>

{

{ "value", "Hello world!" }

};

await host.StartAsync();

await jobHost.CallAsync("CreateQueueMessage", inputs);

await host.StopAsync();

}

}

#### Version 2.x

static void Main(string[] args)

{

JobHost host = new JobHost();

host.Call(typeof(Program).GetMethod("CreateQueueMessage"), new { value = "Hello world!" });

}

## Input and output bindings

Input bindings provide a declarative way to make data from Azure or third-party services available to your code. Output bindings provide a way to update data. The [Get started](https://docs.microsoft.com/en-us/azure/app-service/webjobs-sdk-get-started) article shows an example of each.

You can use a method return value for an output binding by applying the attribute to the method return value. See the example in [Using the Azure Function return value](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-return-value).

**Input binding**

public static class QueueFunctions

{

[FunctionName("QueueTrigger")]

public static void QueueTrigger(

[QueueTrigger("myqueue-items")] string myQueueItem,

ILogger log)

{

log.LogInformation($"C# function processed: {myQueueItem}");

}

}

In above method **QueueTrigger** attribute make available the queue parameter value to method.

Check below link for different resources

[**https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings**](https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings)

**Output Binding**

[FunctionName("QueueTrigger")]

[return: Blob("output-container/{id}")]

public static string Run([QueueTrigger("inputqueue")]WorkItem input, ILogger log)

{

string json = string.Format("{{ \"id\": \"{0}\" }}", input.Id);

log.LogInformation($"C# script processed queue message. Item={json}");

return json;

}

## Binding types

The process for installing and managing binding types depends on whether you're using version 3.x or version 2.x of the SDK. You can find the package to install for a particular binding type in the "Packages" section of that binding type's Azure Functions [reference article](https://docs.microsoft.com/en-us/azure/app-service/webjobs-sdk-how-to#binding-reference-information). An exception is the Files trigger and binding (for the local file system), which isn't supported by Azure Functions.

#### Version 3.x

In version 3.x, the storage bindings are included in the Microsoft.Azure.WebJobs.Extensions.Storage package. Call the AddAzureStorage extension method in the ConfigureWebJobs method, as shown here:

static void Main()

{

var builder = new HostBuilder();

builder.ConfigureWebJobs(b =>

{

b.AddAzureStorageCoreServices();

b.AddAzureStorage();

});

var host = builder.Build();

using (host)

{

host.Run();

}

}

To use other trigger and binding types, install the NuGet package that contains them and call the Add<binding> extension method implemented in the extension. For example, if you want to use an Azure Cosmos DB binding, install Microsoft.Azure.WebJobs.Extensions.CosmosDB and call AddCosmosDB, like this:

static void Main()

{

var builder = new HostBuilder();

builder.ConfigureWebJobs(b =>

{

b.AddAzureStorageCoreServices();

b.AddCosmosDB();

});

var host = builder.Build();

using (host)

{

host.Run();

}

}

To use the Timer trigger or the Files binding, which are part of core services, call the AddTimers or AddFiles extension methods, respectively.

#### Version 2.x

These trigger and binding types are included in version 2.x of the Microsoft.Azure.WebJobs package:

* Blob storage
* Queue storage
* Table storage

To use other trigger and binding types, install the NuGet package that contains them and call a Use<binding> method on the JobHostConfiguration object. For example, if you want to use a Timer trigger, install Microsoft.Azure.WebJobs.Extensions and call UseTimers in the Main method, as shown here:

static void Main()

{

config = new JobHostConfiguration();

config.UseTimers();

var host = new JobHost(config);

host.RunAndBlock();

}

To use the Files binding, install Microsoft.Azure.WebJobs.Extensions and call UseFiles.

### ExecutionContext

WebJobs lets you bind to an [ExecutionContext](https://github.com/Azure/azure-webjobs-sdk-extensions/blob/v2.x/src/WebJobs.Extensions/Extensions/Core/ExecutionContext.cs). With this binding, you can access the [ExecutionContext](https://github.com/Azure/azure-webjobs-sdk-extensions/blob/v2.x/src/WebJobs.Extensions/Extensions/Core/ExecutionContext.cs) as a parameter in your function signature. For example, the following code uses the context object to access the invocation ID, which you can use to correlate all logs produced by a given function invocation.

public class Functions

{

public static void ProcessQueueMessage([QueueTrigger("queue")] string message,

ExecutionContext executionContext,

ILogger logger)

{

logger.LogInformation($"{message}\n{executionContext.InvocationId}");

}

}

The process for binding to the [ExecutionContext](https://github.com/Azure/azure-webjobs-sdk-extensions/blob/v2.x/src/WebJobs.Extensions/Extensions/Core/ExecutionContext.cs) depends on your SDK version.

#### Version 3.x

Call the AddExecutionContextBinding extension method in the ConfigureWebJobs method, as shown here:

static void Main()

{

var builder = new HostBuilder();

builder.ConfigureWebJobs(b =>

{

b.AddAzureStorageCoreServices();

b.AddExecutionContextBinding();

});

var host = builder.Build();

using (host)

{

host.Run();

}

}

#### Version 2.x

The Microsoft.Azure.WebJobs.Extensions package mentioned earlier also provides a special binding type that you can register by calling the UseCore method. This binding lets you define an [ExecutionContext](https://github.com/Azure/azure-webjobs-sdk-extensions/blob/v2.x/src/WebJobs.Extensions/Extensions/Core/ExecutionContext.cs) parameter in your function signature, which is enabled like this:

class Program

{

static void Main()

{

config = new JobHostConfiguration();

config.UseCore();

var host = new JobHost(config);

host.RunAndBlock();

}

}

## Binding configuration

You can configure the behavior of some triggers and bindings. The process for configuring them depends on the SDK version.

* **Version 3.x:** Set configuration when the Add<Binding> method is called in ConfigureWebJobs.
* **Version 2.x:** Set configuration by setting properties in a configuration object that you pass in to JobHost.

These binding-specific settings are equivalent to settings in the [host.json project file](https://docs.microsoft.com/en-us/azure/azure-functions/functions-host-json) in Azure Functions.

You can configure the following bindings:

* [Azure CosmosDB trigger](https://docs.microsoft.com/en-us/azure/app-service/webjobs-sdk-how-to#azure-cosmosdb-trigger-configuration-version-3x)
* [Event Hubs trigger](https://docs.microsoft.com/en-us/azure/app-service/webjobs-sdk-how-to#event-hubs-trigger-configuration-version-3x)
* Queue storage trigger
* [SendGrid binding](https://docs.microsoft.com/en-us/azure/app-service/webjobs-sdk-how-to#sendgrid-binding-configuration-version-3x)
* [Service Bus trigger](https://docs.microsoft.com/en-us/azure/app-service/webjobs-sdk-how-to#service-bus-trigger-configuration-version-3x)

### Azure CosmosDB trigger configuration (version 3.x)

This example shows how to configure the Azure Cosmos DB trigger:

static void Main()

{

var builder = new HostBuilder();

builder.ConfigureWebJobs(b =>

{

b.AddAzureStorageCoreServices();

b.AddCosmosDB(a =>

{

a.ConnectionMode = ConnectionMode.Gateway;

a.Protocol = Protocol.Https;

a.LeaseOptions.LeasePrefix = "prefix1";

});

});

var host = builder.Build();

using (host)

{

host.Run();

}

}

For more details, see the [Azure CosmosDB binding](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-cosmosdb-v2#hostjson-settings) article.

### Event Hubs trigger configuration (version 3.x)

This example shows how to configure the Event Hubs trigger:

static void Main()

{

var builder = new HostBuilder();

builder.ConfigureWebJobs(b =>

{

b.AddAzureStorageCoreServices();

b.AddEventHubs(a =>

{

a.BatchCheckpointFrequency = 5;

a.EventProcessorOptions.MaxBatchSize = 256;

a.EventProcessorOptions.PrefetchCount = 512;

});

});

var host = builder.Build();

using (host)

{

host.Run();

}

}

For more details, see the [Event Hubs binding](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-event-hubs#hostjson-settings) article.

### SendGrid binding configuration (version 3.x)

This example shows how to configure the SendGrid output binding:

static void Main()

{

var builder = new HostBuilder();

builder.ConfigureWebJobs(b =>

{

b.AddAzureStorageCoreServices();

b.AddSendGrid(a =>

{

a.FromAddress.Email = "samples@functions.com";

a.FromAddress.Name = "Azure Functions";

});

});

var host = builder.Build();

using (host)

{

host.Run();

}

}

For more details, see the [SendGrid binding](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-sendgrid#hostjson-settings) article.

### Service Bus trigger configuration (version 3.x)

This example shows how to configure the Service Bus trigger:

static void Main()

{

var builder = new HostBuilder();

builder.ConfigureWebJobs(b =>

{

b.AddAzureStorageCoreServices();

b.AddServiceBus(sbOptions =>

{

sbOptions.MessageHandlerOptions.AutoComplete = true;

sbOptions.MessageHandlerOptions.MaxConcurrentCalls = 16;

});

});

var host = builder.Build();

using (host)

{

host.Run();

}

}

For more details, see the [Service Bus binding](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-service-bus#hostjson-settings) article.

### Queue storage trigger configuration

These examples show how to configure the Queue storage trigger:

#### Version 3.x

static void Main()

{

var builder = new HostBuilder();

builder.ConfigureWebJobs(b =>

{

b.AddAzureStorageCoreServices();

b.AddAzureStorage(a => {

a.BatchSize = 8;

a.NewBatchThreshold = 4;

a.MaxDequeueCount = 4;

a.MaxPollingInterval = TimeSpan.FromSeconds(15);

});

});

var host = builder.Build();

using (host)

{

host.Run();

}

}

For more details, see the [Queue storage binding](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-queue#hostjson-settings) article.

#### Version 2.x

static void Main(string[] args)

{

JobHostConfiguration config = new JobHostConfiguration();

config.Queues.BatchSize = 8;

config.Queues.NewBatchThreshold = 4;

config.Queues.MaxDequeueCount = 4;

config.Queues.MaxPollingInterval = TimeSpan.FromSeconds(15);

JobHost host = new JobHost(config);

host.RunAndBlock();

}

For more details, see the [host.json v1.x reference](https://docs.microsoft.com/en-us/azure/azure-functions/functions-host-json-v1#queues).

### Configuration for other bindings

Some trigger and binding types define their own custom configuration types. For example, the File trigger lets you specify the root path to monitor, as in these examples:

#### Version 3.x

static void Main()

{

var builder = new HostBuilder();

builder.ConfigureWebJobs(b =>

{

b.AddAzureStorageCoreServices();

b.AddFiles(a => a.RootPath = @"c:\data\import");

});

var host = builder.Build();

using (host)

{

host.Run();

}

}

#### Version 2.x

static void Main()

{

config = new JobHostConfiguration();

var filesConfig = new FilesConfiguration

{

RootPath = @"c:\data\import"

};

config.UseFiles(filesConfig);

var host = new JobHost(config);

host.RunAndBlock();

}

## Binding expressions

In attribute constructor parameters, you can use expressions that resolve to values from various sources. For example, in the following code, the path for the BlobTrigger attribute creates an expression named filename. When used for the output binding, filename resolves to the name of the triggering blob.

public static void CreateThumbnail(

[BlobTrigger("sample-images/{filename}")] Stream image,

[Blob("sample-images-sm/{filename}", FileAccess.Write)] Stream imageSmall,

string filename,

ILogger logger)

{

logger.Info($"Blob trigger processing: {filename}");

// ...

}

For more information about binding expressions, see [Binding expressions and patterns](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-expressions-patterns) in the Azure Functions documentation.

### Custom binding expressions

Sometimes you want to specify a queue name, a blob name or container, or a table name in code rather than hard-coding it. For example, you might want to specify the queue name for the QueueTrigger attribute in a configuration file or environment variable.

You can do that by passing a NameResolver object in to the JobHostConfiguration object. You include placeholders in trigger or binding attribute constructor parameters, and your NameResolver code provides the actual values to be used in place of those placeholders. You identify placeholders by surrounding them with percent (%) signs, as shown here:

public static void WriteLog([QueueTrigger("%logqueue%")] string logMessage)

{

Console.WriteLine(logMessage);

}

This code lets you use a queue named logqueuetest in the test environment and one named logqueueprod in production. Instead of a hard-coded queue name, you specify the name of an entry in the appSettings collection.

There's a default NameResolver that takes effect if you don't provide a custom one. The default gets values from app settings or environment variables.

Your NameResolver class gets the queue name from appSettings, as shown here:

public class CustomNameResolver : INameResolver

{

public string Resolve(string name)

{

return ConfigurationManager.AppSettings[name].ToString();

}

}

#### Version 3.x

You configure the resolver by using dependency injection. These samples require the following using statement:

using Microsoft.Extensions.DependencyInjection;

You add the resolver by calling the [ConfigureServices](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.hosting.hostinghostbuilderextensions.configureservices) extension method on [HostBuilder](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.hosting.hostbuilder), as in this example:

static async Task Main(string[] args)

{

var builder = new HostBuilder();

var resolver = new CustomNameResolver();

builder.ConfigureWebJobs(b =>

{

b.AddAzureStorageCoreServices();

});

builder.ConfigureServices(s => s.AddSingleton<INameResolver>(resolver));

var host = builder.Build();

using (host)

{

await host.RunAsync();

}

}

#### Version 2.x

Pass your NameResolver class in to the JobHost object, as shown here:

static void Main(string[] args)

{

JobHostConfiguration config = new JobHostConfiguration();

config.NameResolver = new CustomNameResolver();

JobHost host = new JobHost(config);

host.RunAndBlock();

}

Azure Functions implements INameResolver to get values from app settings, as shown in the example. When you use the WebJobs SDK directly, you can write a custom implementation that gets placeholder replacement values from whatever source you prefer.

## Binding at runtime

If you need to do some work in your function before you use a binding attribute like Queue, Blob, or Table, you can use the IBinder interface.

The following example takes an input queue message and creates a new message with the same content in an output queue. The output queue name is set by code in the body of the function.

public static void CreateQueueMessage(

[QueueTrigger("inputqueue")] string queueMessage,

IBinder binder)

{

string outputQueueName = "outputqueue" + DateTime.Now.Month.ToString();

QueueAttribute queueAttribute = new QueueAttribute(outputQueueName);

CloudQueue outputQueue = binder.Bind<CloudQueue>(queueAttribute);

outputQueue.AddMessageAsync(new CloudQueueMessage(queueMessage));

}

For more information, see [Binding at runtime](https://docs.microsoft.com/en-us/azure/azure-functions/functions-dotnet-class-library#binding-at-runtime) in the Azure Functions documentation.

## Binding reference information

The Azure Functions documentation provides reference information about each binding type. You'll find the following information in each binding reference article. (This example is based on Storage queue.)

* [Packages](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-queue#packages---functions-1x). The package you need to install to include support for the binding in a WebJobs SDK project.
* [Examples](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-queue#trigger---example). Code samples. The C# class library example applies to the WebJobs SDK. Just omit the FunctionName attribute.
* [Attributes](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-queue#trigger---attributes). The attributes to use for the binding type.
* [Configuration](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-queue#trigger---configuration). Explanations of the attribute properties and constructor parameters.
* [Usage](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-queue#trigger---usage). The types you can bind to and information about how the binding works. For example: polling algorithm, poison queue processing.

## Disable attribute

The [Disable](https://github.com/Azure/azure-webjobs-sdk/blob/master/src/Microsoft.Azure.WebJobs/DisableAttribute.cs) attribute lets you control whether a function can be triggered.

In the following example, if the app setting Disable\_TestJob has a value of 1 or True (case insensitive), the function won't run. In that case, the runtime creates a log message Function 'Functions.TestJob' is disabled.

[Disable("Disable\_TestJob")]

public static void TestJob([QueueTrigger("testqueue2")] string message)

{

Console.WriteLine("Function with Disable attribute executed!");

}

When you change app setting values in the Azure portal, the WebJob restarts to pick up the new setting.

The attribute can be declared at the parameter, method, or class level. The setting name can also contain binding expressions.

## Timeout attribute

The [Timeout](https://github.com/Azure/azure-webjobs-sdk/blob/master/src/Microsoft.Azure.WebJobs/TimeoutAttribute.cs) attribute causes a function to be canceled if it doesn't finish within a specified amount of time. In the following example, the function would run for one day without the Timeout attribute. Timeout causes the function to be canceled after 15 seconds.

[Timeout("00:00:15")]

public static async Task TimeoutJob(

[QueueTrigger("testqueue2")] string message,

CancellationToken token,

TextWriter log)

{

await log.WriteLineAsync("Job starting");

await Task.Delay(TimeSpan.FromDays(1), token);

await log.WriteLineAsync("Job completed");

}

You can apply the Timeout attribute at the class or method level, and you can specify a global timeout by using JobHostConfiguration.FunctionTimeout. Class-level or method-level timeouts override global timeouts.

## Singleton attribute

The [Singleton](https://github.com/Azure/azure-webjobs-sdk/blob/master/src/Microsoft.Azure.WebJobs/SingletonAttribute.cs) attribute ensures that only one instance of a function runs, even when there are multiple instances of the host web app. It does this by using [distributed locking](https://docs.microsoft.com/en-us/azure/app-service/webjobs-sdk-how-to#viewing-lease-blobs).

In this example, only a single instance of the ProcessImage function runs at any given time:

[Singleton]

public static async Task ProcessImage([BlobTrigger("images")] Stream image)

{

// Process the image.

}

### SingletonMode.Listener

Some triggers have built-in support for concurrency management:

* **QueueTrigger**. Set JobHostConfiguration.Queues.BatchSize to 1.
* **ServiceBusTrigger**. Set ServiceBusConfiguration.MessageOptions.MaxConcurrentCalls to 1.
* **FileTrigger**. Set FileProcessor.MaxDegreeOfParallelism to 1.

You can use these settings to ensure that your function runs as a singleton on a single instance. To ensure that only a single instance of the function is running when the web app scales out to multiple instances, apply a listener-level singleton lock on the function ([Singleton(Mode = SingletonMode.Listener)]). Listener locks are acquired when the JobHost starts. If three scaled-out instances all start at the same time, only one of the instances acquires the lock and only one listener starts.

### Scope values

You can specify a scope expression/value on a singleton. The expression/value ensures that all executions of the function at a specific scope will be serialized. Implementing more granular locking in this way can allow for some level of parallelism for your function while serializing other invocations as dictated by your requirements. For example, in the following code, the scope expression binds to the Region value of the incoming message. When the queue contains three messages in regions East, East, and West respectively, the messages that have region East are run serially while the message with region West is run in parallel with those in East.

[Singleton("{Region}")]

public static async Task ProcessWorkItem([QueueTrigger("workitems")] WorkItem workItem)

{

// Process the work item.

}

public class WorkItem

{

public int ID { get; set; }

public string Region { get; set; }

public int Category { get; set; }

public string Description { get; set; }

}

### Scope values

You can specify a scope expression/value on a singleton. The expression/value ensures that all executions of the function at a specific scope will be serialized. Implementing more granular locking in this way can allow for some level of parallelism for your function while serializing other invocations as dictated by your requirements. For example, in the following code, the scope expression binds to the Region value of the incoming message. When the queue contains three messages in regions East, East, and West respectively, the messages that have region East are run serially while the message with region West is run in parallel with those in East.

[Singleton("{Region}")]

public static async Task ProcessWorkItem([QueueTrigger("workitems")] WorkItem workItem)

{

// Process the work item.

}

public class WorkItem

{

public int ID { get; set; }

public string Region { get; set; }

public int Category { get; set; }

public string Description { get; set; }

}

### SingletonScope.Host

The default scope for a lock is SingletonScope.Function, meaning the lock scope (the blob lease path) is tied to the fully qualified function name. To lock across functions, specify SingletonScope.Host and use a scope ID name that's the same across all functions that you don't want to run simultaneously. In the following example, only one instance of AddItem or RemoveItem runs at a time:

[Singleton("ItemsLock", SingletonScope.Host)]

public static void AddItem([QueueTrigger("add-item")] string message)

{

// Perform the add operation.

}

[Singleton("ItemsLock", SingletonScope.Host)]

public static void RemoveItem([QueueTrigger("remove-item")] string message)

{

// Perform the remove operation.

}

### Viewing lease blobs

The WebJobs SDK uses [Azure blob leases](https://docs.microsoft.com/en-us/azure/storage/common/storage-concurrency#pessimistic-concurrency-for-blobs) under the covers to implement distributed locking. The lease blobs used by Singleton can be found in the azure-webjobs-host container in the AzureWebJobsStorage storage account under the path "locks". For example, the lease blob path for the first ProcessImage example shown earlier might be locks/061851c758f04938a4426aa9ab3869c0/WebJobs.Functions.ProcessImage. All paths include the JobHost ID, in this case 061851c758f04938a4426aa9ab3869c0.

## Multiple instances

If your web app runs on multiple instances, a continuous WebJob runs on each instance, listening for triggers and calling functions. The various trigger bindings are designed to efficiently share work collaboratively across instances, so that scaling out to more instances allows you to handle more load.

The queue and blob triggers automatically prevent a function from processing a queue message or blob more than once; functions don't have to be idempotent.

The timer trigger automatically ensures that only one instance of the timer runs, so you don't get more than one function instance running at a given scheduled time.

If you want to ensure that only one instance of a function runs even when there are multiple instances of the host web app, you can use the [Singleton](https://docs.microsoft.com/en-us/azure/app-service/webjobs-sdk-how-to#singleton-attribute) attribute.

**Azure Queue storage bindings for Azure Functions**

Azure Functions supports trigger and output bindings for queues.

The following example shows a [C# function](https://docs.microsoft.com/en-us/azure/azure-functions/functions-dotnet-class-library) that polls the myqueue-items queue and writes a log each time a queue item is processed.

public static class QueueFunctions

{

[FunctionName("QueueTrigger")]

public static void QueueTrigger(

[QueueTrigger("myqueue-items")] string myQueueItem,

ILogger log)

{

log.LogInformation($"C# function processed: {myQueueItem}");

}

}

The following example shows a [C# function](https://docs.microsoft.com/en-us/azure/azure-functions/functions-dotnet-class-library) that polls the myqueue-items queue and writes a log each time a queue item is processed.

public static class QueueFunctions

{

[FunctionName("QueueTrigger")]

public static void QueueTrigger(

[QueueTrigger("myqueue-items")] string myQueueItem,

ILogger log)

{

log.LogInformation($"C# function processed: {myQueueItem}");

}

}

### Trigger - C# script example

The following example shows a queue trigger binding in a function.json file and [C# script (.csx)](https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference-csharp) code that uses the binding. The function polls the myqueue-items queue and writes a log each time a queue item is processed.

Here's the function.json file:

{

"disabled": false,

"bindings": [

{

"type": "queueTrigger",

"direction": "in",

"name": "myQueueItem", // Variable name

"queueName": "myqueue-items",

"connection":"MyStorageConnectionAppSetting"

}

]

}

The [configuration](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-queue#trigger---configuration) section explains these properties.

## Trigger - configuration

The following table explains the binding configuration properties that you set in the function.json file and the QueueTrigger attribute.

| **function.json property** | **Attribute property** | **Description** |
| --- | --- | --- |
| **type** | n/a | Must be set to queueTrigger. This property is set automatically when you create the trigger in the Azure portal. |
| **direction** | n/a | In the function.json file only. Must be set to in. This property is set automatically when you create the trigger in the Azure portal. |
| **name** | n/a | The name of the variable that contains the queue item payload in the function code. |
| **queueName** | **QueueName** | The name of the queue to poll. |
| **connection** | **Connection** | The name of an app setting that contains the Storage connection string to use for this binding. If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set connection to "MyStorage", the Functions runtime looks for an app setting that is named "AzureWebJobsMyStorage." If you leave connection empty, the Functions runtime uses the default Storage connection string in the app setting that is named AzureWebJobsStorage. |
|  |  |  |

When you're developing locally, app settings go into the [local.settings.json file](https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local#local-settings-file).

### Trigger - JavaScript example

The following example shows a queue trigger binding in a function.json file and a [JavaScript function](https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference-node) that uses the binding. The function polls the myqueue-items queue and writes a log each time a queue item is processed.

Here's the function.json file:

{

"disabled": false,

"bindings": [

{

"type": "queueTrigger",

"direction": "in",

"name": "myQueueItem",

"queueName": "myqueue-items",

"connection":"MyStorageConnectionAppSetting"

}

]

}

Here's the JavaScript code:

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

context.log('Node.js queue trigger function processed work item', message);

// OR access using context.bindings.<name>

// context.log('Node.js queue trigger function processed work item', context.bindings.myQueueItem);

context.log('expirationTime =', context.bindingData.expirationTime);

context.log('insertionTime =', context.bindingData.insertionTime);

context.log('nextVisibleTime =', context.bindingData.nextVisibleTime);

context.log('id =', context.bindingData.id);

context.log('popReceipt =', context.bindingData.popReceipt);

context.log('dequeueCount =', context.bindingData.dequeueCount);

context.done();

};

## Trigger - attributes

In [C# class libraries](https://docs.microsoft.com/en-us/azure/azure-functions/functions-dotnet-class-library), use the following attributes to configure a queue trigger:

* [QueueTriggerAttribute](https://github.com/Azure/azure-webjobs-sdk/blob/master/src/Microsoft.Azure.WebJobs/QueueTriggerAttribute.cs)

The attribute's constructor takes the name of the queue to monitor, as shown in the following example:

[FunctionName("QueueTrigger")]

public static void Run(

[QueueTrigger("myqueue-items")] string myQueueItem,

ILogger log)

{

...

}

You can set the Connection property to specify the storage account to use, as shown in the following example:

[FunctionName("QueueTrigger")]

public static void Run(

[QueueTrigger("myqueue-items", Connection = "StorageConnectionAppSetting")] string myQueueItem,

ILogger log)

{

....

}

* [StorageAccountAttribute](https://github.com/Azure/azure-webjobs-sdk/blob/master/src/Microsoft.Azure.WebJobs/StorageAccountAttribute.cs)

Provides another way to specify the storage account to use. The constructor takes the name of an app setting that contains a storage connection string. The attribute can be applied at the parameter, method, or class level. The following example shows class level and method level:

[StorageAccount("ClassLevelStorageAppSetting")]

public static class AzureFunctions

{

[FunctionName("QueueTrigger")]

[StorageAccount("FunctionLevelStorageAppSetting")]

public static void Run( //...

{

...

}

The storage account to use is determined in the following order:

* The QueueTrigger attribute's Connection property.
* The StorageAccount attribute applied to the same parameter as the QueueTrigger attribute.
* The StorageAccount attribute applied to the function.
* The StorageAccount attribute applied to the class.
* The "AzureWebJobsStorage" app setting.

## Trigger - poison messages

When a queue trigger function fails, Azure Functions retries the function up to five times for a given queue message, including the first try. If all five attempts fail, the functions runtime adds a message to a queue named <originalqueuename>-poison. You can write a function to process messages from the poison queue by logging them or sending a notification that manual attention is needed.

To handle poison messages manually, check the [dequeueCount](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-queue#trigger---message-metadata) of the queue message.

## Trigger - polling algorithm

The queue trigger implements a random exponential back-off algorithm to reduce the effect of idle-queue polling on storage transaction costs. When a message is found, the runtime waits two seconds and then checks for another message; when no message is found, it waits about four seconds before trying again. After subsequent failed attempts to get a queue message, the wait time continues to increase until it reaches the maximum wait time, which defaults to one minute. The maximum wait time is configurable via the maxPollingInterval property in the [host.json file](https://docs.microsoft.com/en-us/azure/azure-functions/functions-host-json#queues).

## Trigger - concurrency

When there are multiple queue messages waiting, the queue trigger retrieves a batch of messages and invokes function instances concurrently to process them. By default, the batch size is 16. When the number being processed gets down to 8, the runtime gets another batch and starts processing those messages. So the maximum number of concurrent messages being processed per function on one virtual machine (VM) is 24. This limit applies separately to each queue-triggered function on each VM. If your function app scales out to multiple VMs, each VM will wait for triggers and attempt to run functions. For example, if a function app scales out to 3 VMs, the default maximum number of concurrent instances of one queue-triggered function is 72.

The batch size and the threshold for getting a new batch are configurable in the [host.json file](https://docs.microsoft.com/en-us/azure/azure-functions/functions-host-json#queues). If you want to minimize parallel execution for queue-triggered functions in a function app, you can set the batch size to 1. This setting eliminates concurrency only so long as your function app runs on a single virtual machine (VM).

The queue trigger automatically prevents a function from processing a queue message multiple times; functions do not have to be written to be idempotent.

### Output - C# example

The following example shows a [C# function](https://docs.microsoft.com/en-us/azure/azure-functions/functions-dotnet-class-library) that creates a queue message for each HTTP request received.

[StorageAccount("AzureWebJobsStorage")]

public static class QueueFunctions

{

[FunctionName("QueueOutput")]

[return: Queue("myqueue-items")]

public static string QueueOutput([HttpTrigger] dynamic input, ILogger log)

{

log.LogInformation($"C# function processed: {input.Text}");

return input.Text;

}

}

### Output - C# script example

The following example shows an HTTP trigger binding in a function.json file and [C# script (.csx)](https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference-csharp) code that uses the binding. The function creates a queue item with a **CustomQueueMessage** object payload for each HTTP request received.

**Azure Blob storage bindings for Azure Functions**

## Trigger

The Blob storage trigger starts a function when a new or updated blob is detected. The blob contents are provided as input to the function.

The [Event Grid trigger](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-event-grid) has built-in support for [blob events](https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blob-event-overview) and can also be used to start a function when a new or updated blob is detected. For an example, see the [Image resize with Event Grid](https://docs.microsoft.com/en-us/azure/event-grid/resize-images-on-storage-blob-upload-event) tutorial.

Use Event Grid instead of the Blob storage trigger for the following scenarios:

* Blob storage accounts
* High scale
* Minimizing latency

### Blob storage accounts

[Blob storage accounts](https://docs.microsoft.com/en-us/azure/storage/common/storage-account-overview#types-of-storage-accounts) are supported for blob input and output bindings but not for blob triggers. Blob storage triggers require a general-purpose storage account.

### High scale

High scale can be loosely defined as containers that have more than 100,000 blobs in them or storage accounts that have more than 100 blob updates per second.

### Latency issues

If your function app is on the Consumption plan, there can be up to a 10-minute delay in processing new blobs if a function app has gone idle. To avoid this latency, you can switch to an App Service plan with Always On enabled. You can also use an [Event Grid trigger](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-event-grid) with your Blob storage account. For an example, see the [Event Grid tutorial](https://docs.microsoft.com/en-us/azure/event-grid/resize-images-on-storage-blob-upload-event?toc=%2Fazure%2Fazure-functions%2Ftoc.json).

### Trigger - C# example

The following example shows a [C# function](https://docs.microsoft.com/en-us/azure/azure-functions/functions-dotnet-class-library) that writes a log when a blob is added or updated in the samples-workitems container.

[FunctionName("BlobTriggerCSharp")]

public static void Run([BlobTrigger("samples-workitems/{name}")] Stream myBlob, string name, ILogger log)

{

log.LogInformation($"C# Blob trigger function Processed blob\n Name:{name} \n Size: {myBlob.Length} Bytes");

}

The string {name} in the blob trigger path samples-workitems/{name} creates a [binding expression](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-expressions-patterns) that you can use in function code to access the file name of the triggering blob. For more information, see [Blob name patterns](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob#trigger---blob-name-patterns) later in this article.

For more information about the BlobTrigger attribute, see [Trigger - attributes](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob#trigger---attributes).

### Trigger - C# script example

The following example shows a blob trigger binding in a function.json file and [Python code](https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference-python) that uses the binding. The function writes a log when a blob is added or updated in the samples-workitems [container](https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction#blob-storage-resources).

Here's the binding data in the function.json file:

{

"disabled": false,

"bindings": [

{

"name": "myBlob",

"type": "blobTrigger",

"direction": "in",

"path": "samples-workitems/{name}",

"connection":"MyStorageAccountAppSetting"

}

]

}

The string {name} in the blob trigger path samples-workitems/{name} creates a [binding expression](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-expressions-patterns) that you can use in function code to access the file name of the triggering blob. For more information, see [Blob name patterns](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob#trigger---blob-name-patterns) later in this article.

For more information about function.json file properties, see the [Configuration](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob#trigger---configuration) section explains these properties.

Here's C# script code that binds to a Stream:

public static void Run(Stream myBlob, string name, ILogger log)

{

log.LogInformation($"C# Blob trigger function Processed blob\n Name:{name} \n Size: {myBlob.Length} Bytes");

}

Here's C# script code that binds to a CloudBlockBlob:

#r "Microsoft.WindowsAzure.Storage"

using Microsoft.WindowsAzure.Storage.Blob;

public static void Run(CloudBlockBlob myBlob, string name, ILogger log)

{

log.LogInformation($"C# Blob trigger function Processed blob\n Name:{name}\nURI:{myBlob.StorageUri}");

}

### Trigger - JavaScript example

The following example shows a blob trigger binding in a function.json file and [JavaScript code](https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference-node) that uses the binding. The function writes a log when a blob is added or updated in the samples-workitems container.

Here's the function.json file:

{

"disabled": false,

"bindings": [

{

"name": "myBlob",

"type": "blobTrigger",

"direction": "in",

"path": "samples-workitems/{name}",

"connection":"MyStorageAccountAppSetting"

}

]

}

The string {name} in the blob trigger path samples-workitems/{name} creates a [binding expression](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-expressions-patterns) that you can use in function code to access the file name of the triggering blob. For more information, see [Blob name patterns](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob#trigger---blob-name-patterns) later in this article.

For more information about function.json file properties, see the [Configuration](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob#trigger---configuration) section explains these properties.

Here's the JavaScript code:

module.exports = function(context) {

context.log('Node.js Blob trigger function processed', context.bindings.myBlob);

context.done();

};

## Trigger - attributes

In [C# class libraries](https://docs.microsoft.com/en-us/azure/azure-functions/functions-dotnet-class-library), use the following attributes to configure a blob trigger:

* [BlobTriggerAttribute](https://github.com/Azure/azure-webjobs-sdk/blob/master/src/Microsoft.Azure.WebJobs/BlobTriggerAttribute.cs)

The attribute's constructor takes a path string that indicates the container to watch and optionally a [blob name pattern](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob#trigger---blob-name-patterns). Here's an example:

[FunctionName("ResizeImage")]

public static void Run(

[BlobTrigger("sample-images/{name}")] Stream image,

[Blob("sample-images-md/{name}", FileAccess.Write)] Stream imageSmall)

{

....

}

You can set the Connection property to specify the storage account to use, as shown in the following example:

[FunctionName("ResizeImage")]

public static void Run(

[BlobTrigger("sample-images/{name}", Connection = "StorageConnectionAppSetting")] Stream image,

[Blob("sample-images-md/{name}", FileAccess.Write)] Stream imageSmall)

{

....

}

[StorageAccountAttribute](https://github.com/Azure/azure-webjobs-sdk/blob/master/src/Microsoft.Azure.WebJobs/StorageAccountAttribute.cs)

Provides another way to specify the storage account to use. The constructor takes the name of an app setting that contains a storage connection string. The attribute can be applied at the parameter, method, or class level. The following example shows class level and method level:

[StorageAccount("ClassLevelStorageAppSetting")]

public static class AzureFunctions

{

[FunctionName("BlobTrigger")]

[StorageAccount("FunctionLevelStorageAppSetting")]

public static void Run( //...

{

....

}

The storage account to use is determined in the following order:

* The BlobTrigger attribute's Connection property.
* The StorageAccount attribute applied to the same parameter as the BlobTrigger attribute.
* The StorageAccount attribute applied to the function.
* The StorageAccount attribute applied to the class.
* The default storage account for the function app ("AzureWebJobsStorage" app setting).

## Trigger - configuration

The following table explains the binding configuration properties that you set in the function.json file and the BlobTrigger attribute.

| **function.json property** | **Attribute property** | **Description** |
| --- | --- | --- |
| **type** | n/a | Must be set to blobTrigger. This property is set automatically when you create the trigger in the Azure portal. |
| **direction** | n/a | Must be set to in. This property is set automatically when you create the trigger in the Azure portal. Exceptions are noted in the [usage](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob#trigger---usage) section. |
| **name** | n/a | The name of the variable that represents the blob in function code. |
| **path** | **BlobPath** | The [container](https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction#blob-storage-resources) to monitor. May be a [blob name pattern](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob#trigger---blob-name-patterns). |
| **connection** | **Connection** | The name of an app setting that contains the Storage connection string to use for this binding. If the app setting name begins with "AzureWebJobs", you can specify only the remainder of the name here. For example, if you set connection to "MyStorage", the Functions runtime looks for an app setting that is named "AzureWebJobsMyStorage." If you leave connection empty, the Functions runtime uses the default Storage connection string in the app setting that is named AzureWebJobsStorage.  The connection string must be for a general-purpose storage account, not a [Blob storage account](https://docs.microsoft.com/en-us/azure/storage/common/storage-account-overview#types-of-storage-accounts). |

## Trigger - usage

In C# and C# script, you can use the following parameter types for the triggering blob:

* Stream
* TextReader
* string
* Byte[]
* A POCO serializable as JSON
* ICloudBlob1
* CloudBlockBlob1
* CloudPageBlob1
* CloudAppendBlob1

1 Requires "inout" binding direction in function.json or FileAccess.ReadWrite in a C# class library.

If you try to bind to one of the Storage SDK types and get an error message, make sure that you have a reference to [the correct Storage SDK version](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob#azure-storage-sdk-version-in-functions-1x).

Binding to string, Byte[], or POCO is only recommended if the blob size is small, as the entire blob contents are loaded into memory. Generally, it is preferable to use a Stream or CloudBlockBlob type. For more information, see [Concurrency and memory usage](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob#trigger---concurrency-and-memory-usage) later in this article.

In JavaScript, access the input blob data using context.bindings.<name from function.json>.

## Trigger - blob name patterns

You can specify a blob name pattern in the path property in function.json or in the BlobTrigger attribute constructor. The name pattern can be a [filter or binding expression](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-expressions-patterns). The following sections provide examples.

### Get file name and extension

The following example shows how to bind to the blob file name and extension separately:

"path": "input/{blobname}.{blobextension}",

If the blob is named original-Blob1.txt, the values of the blobname and blobextension variables in function code are original-Blob1 and txt.

### Filter on blob name

The following example triggers only on blobs in the input container that start with the string "original-":

"path": "input/original-{name}",

If the blob name is original-Blob1.txt, the value of the name variable in function code is Blob1.

### Filter on file type

The following example triggers only on .png files:

"path": "samples/{name}.png",

### Filter on curly braces in file names

To look for curly braces in file names, escape the braces by using two braces. The following example filters for blobs that have curly braces in the name:

"path": "images/{{20140101}}-{name}",

If the blob is named {20140101}-soundfile.mp3, the name variable value in the function code is soundfile.mp3.

## Trigger - blob receipts

The Azure Functions runtime ensures that no blob trigger function gets called more than once for the same new or updated blob. To determine if a given blob version has been processed, it maintains blob receipts.

Azure Functions stores blob receipts in a container named azure-webjobs-hosts in the Azure storage account for your function app (defined by the app setting AzureWebJobsStorage). A blob receipt has the following information:

* The triggered function ("<function app name>.Functions.<function name>", for example: "MyFunctionApp.Functions.CopyBlob")
* The container name
* The blob type ("BlockBlob" or "PageBlob")
* The blob name
* The ETag (a blob version identifier, for example: "0x8D1DC6E70A277EF")

To force reprocessing of a blob, delete the blob receipt for that blob from the azure-webjobs-hosts container manually. While reprocessing might not occur immediately, it's guaranteed to occur at a later point in time.

## Trigger - poison blobs

When a blob trigger function fails for a given blob, Azure Functions retries that function a total of 5 times by default.

If all 5 tries fail, Azure Functions adds a message to a Storage queue named webjobs-blobtrigger-poison. The queue message for poison blobs is a JSON object that contains the following properties:

* FunctionId (in the format <function app name>.Functions.<function name>)
* BlobType ("BlockBlob" or "PageBlob")
* ContainerName
* BlobName
* ETag (a blob version identifier, for example: "0x8D1DC6E70A277EF")

## Trigger - concurrency and memory usage

The blob trigger uses a queue internally, so the maximum number of concurrent function invocations is controlled by the [queues configuration in host.json](https://docs.microsoft.com/en-us/azure/azure-functions/functions-host-json#queues). The default settings limit concurrency to 24 invocations. This limit applies separately to each function that uses a blob trigger.

[The consumption plan](https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale#how-the-consumption-and-premium-plans-work) limits a function app on one virtual machine (VM) to 1.5 GB of memory. Memory is used by each concurrently executing function instance and by the Functions runtime itself. If a blob-triggered function loads the entire blob into memory, the maximum memory used by that function just for blobs is 24 \* maximum blob size. For example, a function app with three blob-triggered functions and the default settings would have a maximum per-VM concurrency of 3\*24 = 72 function invocations.

JavaScript and Java functions load the entire blob into memory, and C# functions do that if you bind to string, Byte[], or POCO.

## Trigger - polling

If the blob container being monitored contains more than 10,000 blobs (across all containers), the Functions runtime scans log files to watch for new or changed blobs. This process can result in delays. A function might not get triggered until several minutes or longer after the blob is created.